Machine Learning Approaches to Predicting StepMania Chart Difficulty

Jeroen Meringa, Yoey Tolboom

February 7, 2024

Department of Mathematics
Utrecht University

Abstract

This report presents a comprehensive exploration of machine learning models for predicting the difficulty of StepMania game charts. We compare the performance of standard linear regression, decision tree, random forest regression, random forest classifier, and neural network models. Our findings include insights into the predictive capabilities of each model, with the Neural Network consistently outperforming others. We examine limitations and potential improvements in feature engineering, time series analysis, and model optimization. Path signatures, ARIMA, ARCH, GARCH, and State Space Models are considered for future enhancements.

Contents

1 Introduction **3**

2 General Methodology **4**
2.1 The Database 4
2.2 Parsing Data 5
2.3 Summary of Used Models and Programming Languages 8

3 Initial Models **8**
3.1 Standard Linear Regression 8
3.1.1 Initial Model 8
3.1.2 Initial Results 9
3.2 Decision Tree 11
3.2.1 Initial Model 11
3.2.2 Initial Results 12
3.3 Random Forest Regression 13
3.3.1 Initial Model 14
3.3.2 Initial Results 15
3.4 Random Forest Classifier 18
3.4.1 Initial Model 18
3.4.2 Initial Results 19
3.5 Neural Network 22
3.5.1 Initial Model 23
3.5.2 Initial Results 26

4 Initial Model Insights for Improvements **27**

5 Improved Models **29**
5.1 Standard Linear Regression 29
5.1.1 Improved Model 29
5.1.2 Improved Model Results 30
5.2 Decision Tree 32
5.2.1 Improved Model 32
5.2.2 Improved Model Results 33
5.3 Random Forest Regression 35
5.3.1 Improved Model 35
5.3.2 Improved Model Results 36
5.4 Random Forest Classifier 39
5.4.1 Improved Model 39
5.4.2 Improved Model Results 40
5.5 Neural Network 44
5.5.1 Improved Model 45
5.5.2 Improved Model Results 45

6 Comparing Performance & Discussion **47**

7 Conclusion **48**

8 Recommendations **48**

1 Introduction

Dance Dance Revolution (DDR) stands as a popular rhythm-based dance simulation game that got known worldwide since its inception in the late 1990s. The game challenges players to synchronize their steps with on-screen cues, typically done on an arcade machine, although many home-version varieties exist. Eventually, many spin-off games were created, such as StepMania which we will refer to.

Typical Dance Dance Revolution arcade machines.

The goal of the game is simply to step on the correct panel at the right time, usually in sync with the music. Missing too many arrows results in failing a song, while high accuracy and hitting many arrows is rewarded.

Arrows scroll from the bottom to the top of the screen, while players have to step on the corresponding machine panel when the arrows overlap the indicator (receptor) at the top of the screen. A set of instruction-arrows corresponding to a specific song is called a “stepchart” or simply “chart” for short.

The game originally provided a diverse array of songs and charts, ranging from beginner to expert levels. The easiest charts were simply given a difficulty of “1”, and the hardest ones went up to 10. The game inspired many spin-offs (In The Groove, Stepmania and many more titles) and as enthusiasts continually looked for new ways to engage with the game, eventually custom software for players to create their own charts was developed. This resulted in higher and higher difficulties of charts being created to this day.
However, there are no rules for how difficulties are assigned to charts. Intuitively, more and faster steps or more complex patterns contribute to higher difficulties. If a chart is more exhausting and difficult to play than any of the level 10’s out there, then surely it is at least a

level 11 right? As in the modern game difficulties over 30 now exist, understanding and predicting the difficulty of StepMania charts becomes an interesting aspect for both players and content creators. With tens of thousands of charts being created over the years, this resulted in an interesting database very suitable for machine learning purposes. After all, charts are in essence time-images of instructions, manually labeled by humans based on how they subjectively think is correct. This inadvertently created a fantastic playground data-set for the development of machine leaning methods, assessing chart difficulties. Especially because the difficulties can be considered as sequential categories, making it suitable for both regression and classification (and hybrid) approaches. Not only that, but charts are essentially time series of instructions, opening the door for all sorts of time series analyses as well. Not only that, but charts can be parsed in many different ways. Often, the input space of models is predefined by the data, but in this case, the data is parsed from the game files. This means that the explanatory variable space can be created freely as seen fit. It makes not only a great tool for scientific learning, but also holds practical implications for the gaming community as difficulties are often disputed since they are initially subjectively assigned by the creator. Objective difficulty rating standards would be a great asset for the competitive scene.
In this report, we build and test several machine learning models and compare their performance. It is not meant to provide an objective rating system, but rather to research whether such approach could contribute in a meaningful way. We'll simply be touching on some of the topics, and don't mean to be thorough or complete. The entire project code is available on GitHub[1].

2 General Methodology

The methodology followed for the different models is in general largely identical. First, the data set is parsed to suit a specific model's requirements. Next, we build an initial configuration of the model and finally we try to improve these models based on the initial results. The performances of each final, improved model will then be compared. We will put effort towards keeping the models interpretable, as machine learning algorithms can quickly become black-boxes where only input and output is known, but the inner workings are difficult to retrace.

2.1 The Database

The database consists of a collection of 15.550 charts, released over approximately the past 15 years. These are .sm or .ssc files containing instructions on when and where a step occurs, as well as some meta-data such as song title, game graphics and song speed in beats per minute (BPM). Not all available charts over that time period have been included, as they would use different rating systems, be of poor quality in general or are intended as "joke" charts which all would negatively impact data quality. We have stripped the data from all game-related assets such as audio files and graphics, as these are irrelevant for the difficulty of a chart and would bloat the database with over 100GB of data.

[1] https://github.com/Jerros/Programing-for-Math-Assignment

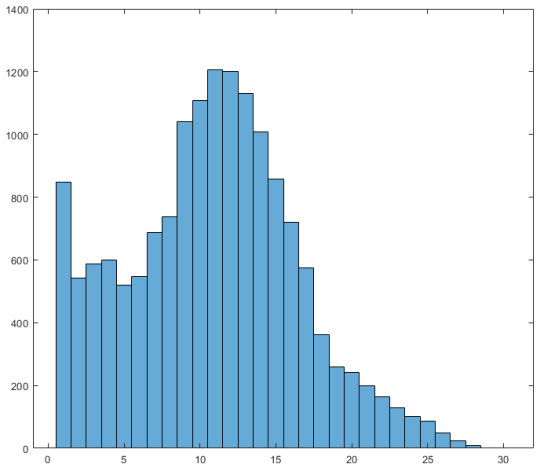


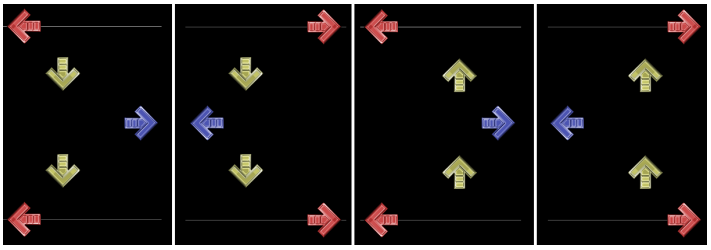
Figure 1: Number of charts per difficulty in the database.

2.2 Parsing Data

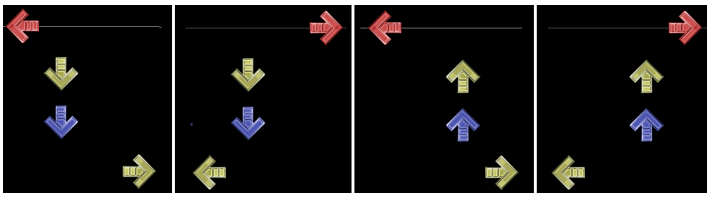
In order to parse the database of .sm and .ssc charts, we used a custom modified version of the program called "ArrowVortex" (AV)[2] as a base, and wrote custom Lua code[3] for each model-specific needs. We chose to use AV as it has built-in object oriented functionalities tailored for .sm and .ssc files, letting us easily use readily available objects and methods relevant to parsing charts. Initially we defined and extracted 29 features per chart we ought to be relevant for the difficulty of a chart to using this as a base case for all models, before fine-tuning them and resort to model-specific parsing later on. These features can be roughly split in 4 groups: the length and amount of steps in a chart, the occurrence of difficult-to-step patterns, how fast the fast parts are and how much of the chart can be considered a res tor break. Some of the most occurring hard-to-step patterns are nicknamed foot-switches, crossovers and afronovers.

[2]ArrowVortex is the main software used worldwide for creating and editing charts, initially created in 2017 by Bram van de Wetering. He has been kind enough to send us use a modified version of the program so we could write our own code for it. See also: https://arrowvortex.ddrnl.com/changelogs.html

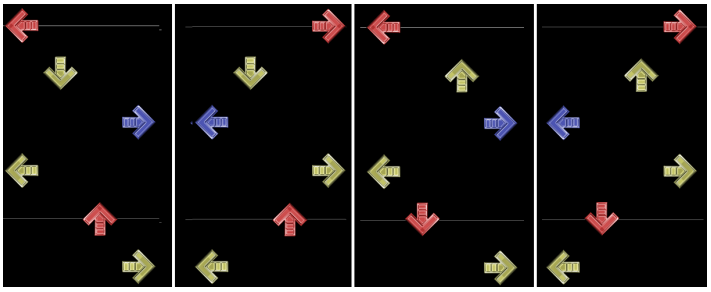
[3]See the main/AV Beta/scripts/ folder of the git project for the parsing code.



(a) Crossover patterns.



(b) Footswitch patterns.



(c) Afronover patterns.

Figure 2: Examples of some technical patterns (to be read from top to bottom). There are experienced as more difficult, as simply stepping such patterns with alternating left and right feet require more body movement. Ideally a player would step the first (top) arrow with either their left or right foot, and then alternate between left and right for each step after (without using the same foot twice in succession). These patterns make such basic movement more difficult and as such are counted and included as input features for the models.

Table 1: Initial Model Features

Name	Description	Expected Impact
Song Length	Length of the chart in seconds	The longer a chart, the harder.
AvgNPS	Average steps per second	Stepping faster is more exhausting.
TotalSteps	Number of steps	More steps increase difficulty.
Crossovers	Number of crossovers	Difficult stepping patterns make for a more difficult chart.
Afronovers	Number of afronovers	
Foot-Switches	Number of footswitches	
TotalTech	Sum of crossovers, afronovers and footswitches	
Jumps	Number of 2 simultaneous steps	
Hands	Number of 3 simultaneous steps	
Quads	Number of 4 simultaneous steps	
TotalAllJumps	Sum of jumps, hands and quads	
RegionNPSprctles[1]	Average steps per second of the 10% percentile fastest region	If the difference between low and high percentile regions is small, that means the entire chart is stepped consistently at the same speed.
RegionNPSprctles[2]	Average steps per second of the 20% percentile fastest region	
RegionNPSprctles[3]	Average steps per second of the 30% percentile fastest region	
RegionNPSprctles[4]	Average steps per second of the 40% percentile fastest region	
RegionNPSprctles[5]	Average steps per second of the 50% percentile fastest region	The difference between median and top percentiles can indicate whether the chart has short, fast bursts, or is more even and consistent in its speed.
RegionNPSprctles[6]	Average steps per second of the 60% percentile fastest region	
RegionNPSprctles[7]	Average steps per second of the 70% percentile fastest region	
RegionNPSprctles[8]	Average steps per second of the 80% percentile fastest region	
RegionNPSprctles[9]	Average steps per second of the 90% percentile fastest region	Faster top speeds contribute to a higher difficulty.
RestRegionsPrct[1]	Percentage of chart slower than 75% of RegionNPSprctles[1]	If part of the chart is relatively slow, it can beconsidered a break, decreasing difficulty.
RestRegionsPrct[2]	Percentage of chart slower than 75% of RegionNPSprctles[2]	
RestRegionsPrct[3]	Percentage of chart slower than 75% of RegionNPSprctles[3]	
RestRegionsPrct[4]	Percentage of chart slower than 75% of RegionNPSprctles[4]	
RestRegionsPrct[5]	Percentage of chart slower than 75% of RegionNPSprctles[5]	
RestRegionsPrct[6]	Percentage of chart slower than 75% of RegionNPSprctles[6]	
RestRegionsPrct[7]	Percentage of chart slower than 75% of RegionNPSprctles[7]	
RestRegionsPrct[8]	Percentage of chart slower than 75% of RegionNPSprctles[8]	
RestRegionsPrct[9]	Percentage of chart slower than 75% of RegionNPSprctles[9]	

All models are initially ran involving these features as a starting point unless specified otherwise.

2.3 Summary of Used Models and Programming Languages

We will look at five different machine learning models. First, a standard linear regression is done, built using MatLab. Second, a decision tree is build in MatLab, taking advantage of the output being sequential categories. This model will be a regression and classification hybrid. Third, we expand the decision tree idea into a random forest regression model. Python in combination with the scikit-learn packages are used for this, and for the fourth model: random forest classification. Fifth and finally, a neural network is written in the C programming language.

3 Initial Models

3.1 Standard Linear Regression

We start with one of the simplest models: linear regression. we use linear regression to fit a predictive model to our observed response and explanatory variables. This can be written in matrix notation as:

$$y = X\beta + \epsilon$$

where y is a vector of observed values, X is a matrix of explanatory variables (including a column of constants), β is a vector of regression coefficients and ϵ a vector of error terms.

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\mathsf{T} \\ \mathbf{x}_2^\mathsf{T} \\ \vdots \\ \mathbf{x}_n^\mathsf{T} \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1p} \\ 1 & x_{21} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{np} \end{bmatrix}$$

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{bmatrix} \qquad \boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix}$$

We then try to minimize the error term using the ordinary least squares (OLS) method, which minimizes the sum of square residuals and leads to a closed-form expression for the estimated value of the unknown parameter vector β:

$$\beta = (X^t X)^{-1} X^T y$$

3.1.1 Initial Model

MatLab was used for the linear regression model, using predictive variables as described in Table 1, with a few changes: we added a constant for the intercept term, and removed the "TotalTech" and "TotalAllJumps" features as those are linear combinations of other explanatory

variables. The training data is a randomized selection of 90% of the database, leaving 10% for testing purposes.[4] Linear regression is then ran on the training data, using heteroskedasticity and autocorrelation corrected standard errors.

3.1.2 Initial Results

After getting the β vector from the model, we use the test set to see its initial performance:

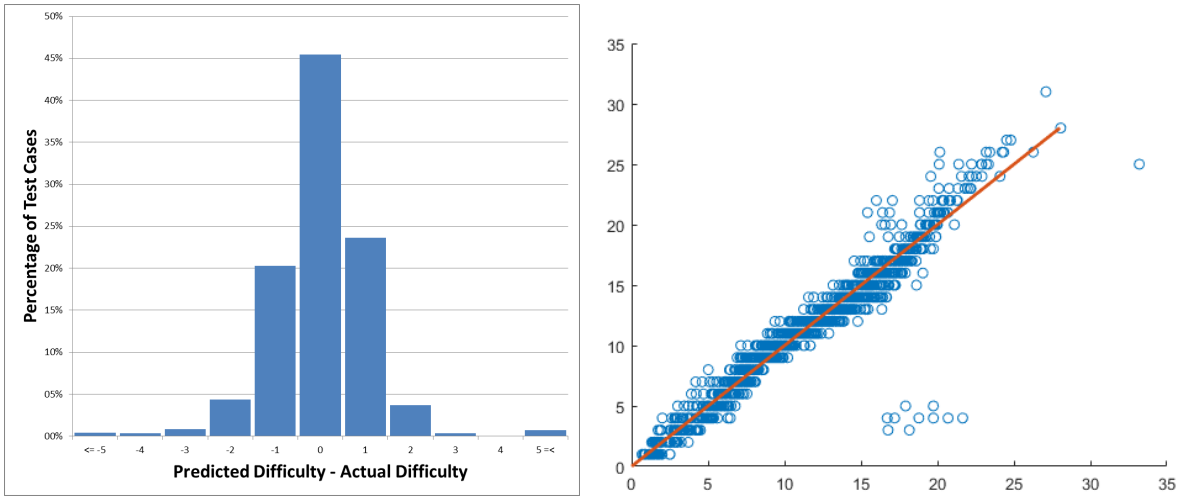


Figure 3: Histogram depicting the prediction accuracy of the initial linear regression model.

The buckets are defined with an interval of ± 0.5 centered around the test rating. In total, 89.3% is identified correctly within a margin of 1, with 45.4% matching perfectly. In the scatterplot, we can see a cluster of charts in the bottom center that is predicted way off. Manual inspection of these charts could perhaps show a common characteristic that the current model is not picking up on.
With linear regression models, it is possible to calculate some meta-data regarding the performance of the model:

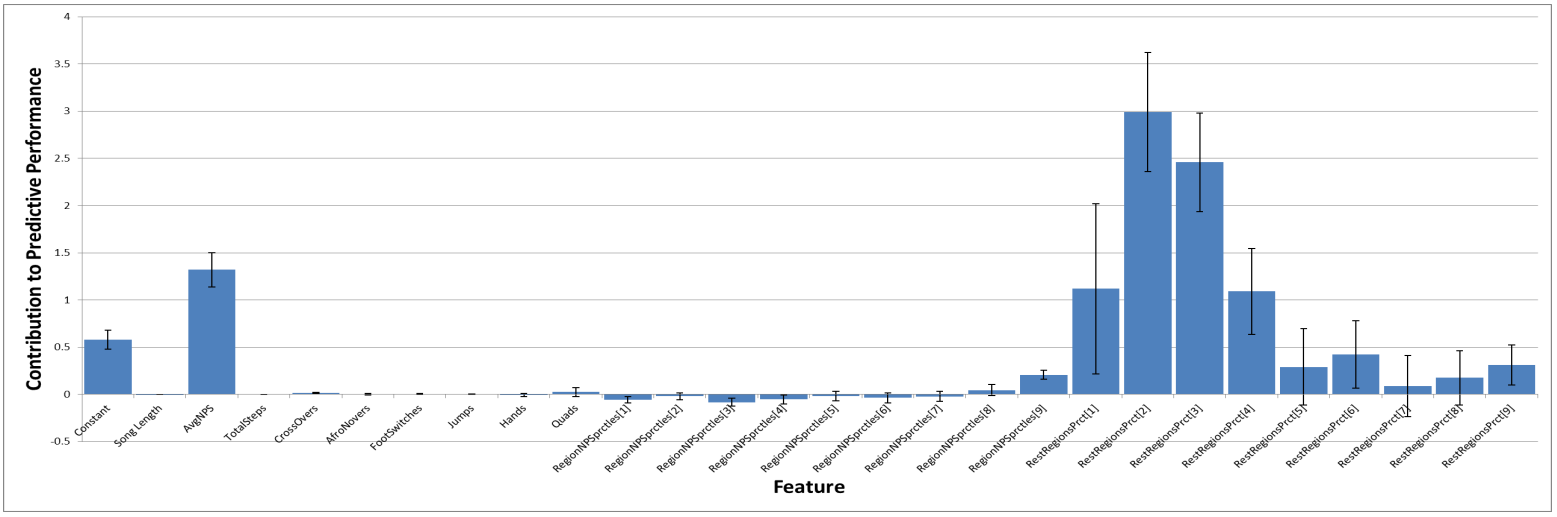
Table 2: Meta statistics about the initial linear regression model.

Statistic	Value
R^2	0.93
F-Stat	6759
P-value	0
estimate of error variance	2.23

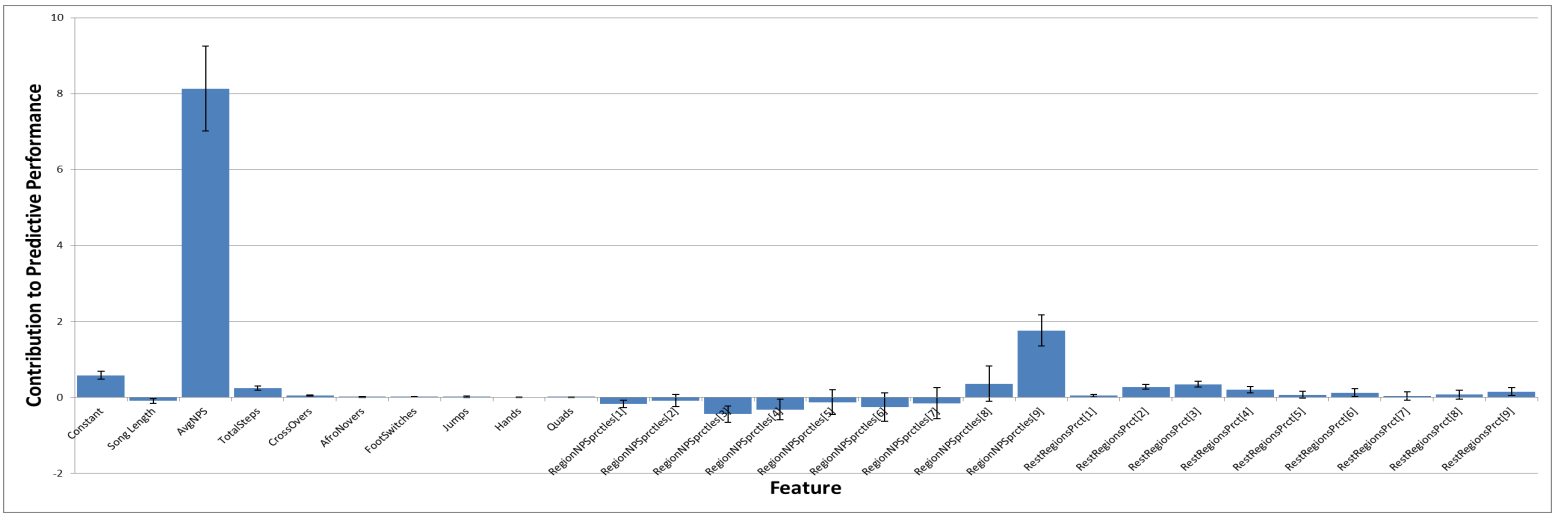
This looks promising at first, with a decent R^2. The predictive capability seems decent as well, so let's look at the weights in the β array, normalized by multiplying the average feature value with β. Normalization is necessary, as otherwise the feature values will seem distorted as seen in figure 4.

[4]For the code, see main/Linear Regression/Create_TrainingTest_Data.m in the git project.

Figure 4: Feature weights (β) of the initial linear regression model, within 95% certainty.



(a) Not normalized, it shows a distorted sense of weights.



(b) Normalized feature weights (β).

The model picks up on the average density of the chart as an important explanatory variable, but other than that it seems to perform questionable. It is expected that the faster percentile regions would contribute heavily positive, and for the rest regions to contribute negatively. But the opposite seems to be the case, and instead the more breaks a chart has, the higher the model estimates the difficulty. It seems as if the model uses the average and top density region as the most important explanatory variables, and then compensates with how fast the slower regions are, although error margins are large. Note that the intersect term seems to be about 0.5, which could be explained by how the target values are rounded down in a sense. For example, a 14.9 would be considered a "difficult 14" and not a 15 (the difficulty is assigned as a floor function in a sense). That could explain the constant term of 0.5. We also note the mistake of including all of the fast and rest regions. As these are expressed as a percentage of the chart above or below a certain density threshold, these percentages can add up to 1, letting linear combinations of columns be equal to others. This should be avoided when doing linear regression. All in all though, the high accuracy combined with poor performance regarding attributing weights to the correct features, could be an indication of over-fitting

and the model having too many variables. We will test and address this when improving the model further. But first, we'll take a look at the initial performance of the other models for more insight regarding possible optimizations.

3.2 Decision Tree

Decision tree learning is a supervised learning approach using a tree-like structure, where each internal node represents a decision based on the value of a particular feature. Each branch represents an outcome of that decision, and each leaf node represents the predicted outcome.

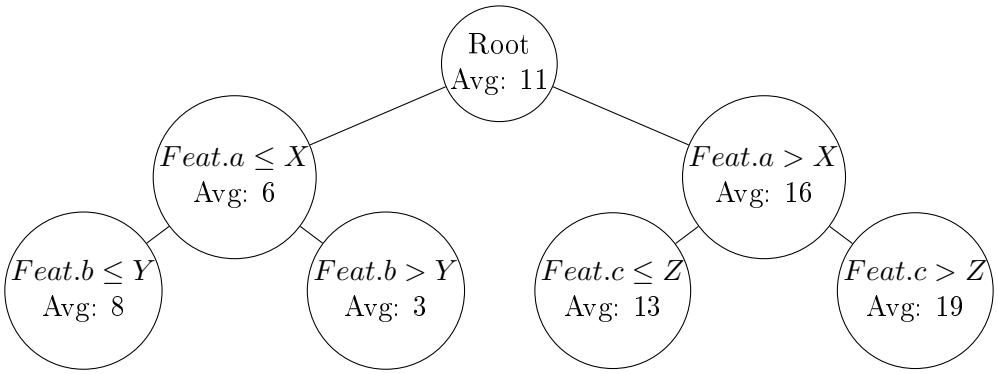


Figure 5: Example of a decision tree. At each node, a feature and feature value are selected such that the population is split into two sub-populations that have the least sum of squares to their average value.

Tree models where the target variable has discrete sets of values are called classification trees, where the leaves represent class labels and branches represent conjunctions of features that lead to those class labels. Decision trees where the target variable is continuous are regression trees. But the concept can be extended to any kind of object equipped with pairwise dissimilarities[5], like categorical sequences which we are working with.

3.2.1 Initial Model

We use MatLab for the decision tree model, with the following initial specifications:

Table 3: Initial Model Parameters

Parameter	Value	Clarification
MaxTreeLength	21	Maximum depth of the tree.
NSteps	1000	The granularity of which feature values are picked for splitting: $\frac{Max\_ft.val - Min\_ft.val}{NSteps}$.
MinLeaves	2	Branch won't be split if the number of samples $\leq$ MinLeaves.

The initial decision tree is built with these parameters.

We then built the decision tree model in MatLab. The training data is a randomized selection of 90% of the database, leaving 10% for testing purposes.[6]

[5] Studer, Matthias; Ritschard, Gilbert; Gabadinho, Alexis; Müller, Nicolas S. (2011). "Discrepancy Analysis of State Sequences"

[6] For the code, see main/Decision Tree/Plant_DesTree.m in the git project.

The main outline of the algorithm is as follows:

- From the first node containing the entire training database of samples, start with the first feature and divide the difference between the highest and lowest feature value by the granularity parameter: $\frac{Max\_ft.val - Min\_ft.val}{NSteps}$. This gives us $NSteps$ feature value cutoffs.
- Calculate the sum of square residuals (SSR) of the response data from each group to their population mean. This leave us with one SSR for each group. The total SSR of the currently inspected feature (f) and split value (v) is simply:

$$SSR_{f,v} = SSR_{f,v,1} + SSR_{f,v,2}.$$

- Repeat this process for all features and split values , and find the best possible combination to split a node, defined by the lowest value out of all $SSR_{f,v}$.
- Split all samples of the node in two groups, with the first group containing all samples with feature values $\leq$ the chosen cutoff, and the remainder in the second group.
- Repeat all steps for the two newly created nodes, until we are either $MaxTreeLength$ branches deep, each leave contains $MinLeaves$ or fewer samples or all leaves have the same prediction value.

This will then recursively build a decision tree. This model can then be fed an input chart, which will flow down the tree according to its features until it arrives at a final nodes. The mean of all output values of the leaves in that node will then be the predicted output for this chart.

3.2.2 Initial Results

We now run our test set through the decision tree to see its initial performance:

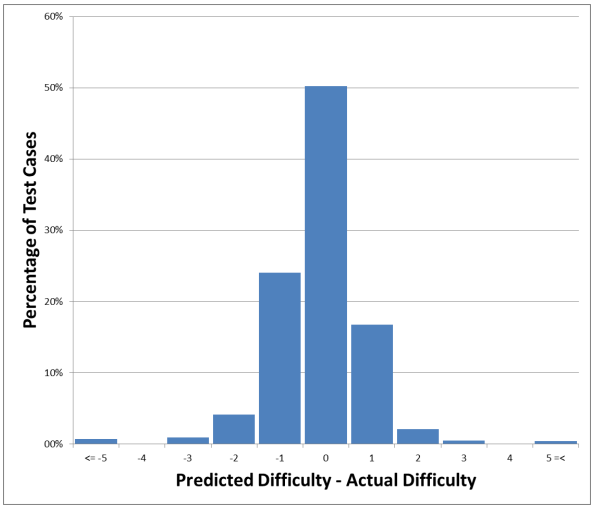


Figure 6: Histogram depicting the prediction accuracy of the decision tree model.

The buckets are defined with an interval of ± 0.5 centered around the test rating. In total, 91.1% is identified correctly within a margin of 1.
Using a decision tree like this has several advantages. First, the hybrid classification-regression approach we use leads to nodes that can consist of different target outputs. This is usually

sub-optimal and considered as some noise, but we can use this to our advantage. For example, if a final branch has 8 leaves with 4 of difficulty 14 and 4 of difficulty 15, then we can simply take the average of all leaves like a regression tree as our output. This is normally not possible in a classifier model, but we have sequential classes so it's possible to average them out. This also provides some extra insight - seemingly the model couldn't split such branch any further with a lower SSR so the leaves are all very similar. If target 14 and 15 are thus all in the same branch, we can interpret those as "hard 14" or "easy 15", and assign it a difficulty of $14,5$. In especially classifier models, this information could get lost and the test case would simply be identified as a 14 or 15.
Another advantage, is that the model tells us exactly how it goes about assigning output values to test cases. A lot of machine learning models might function as a black-box, where much of the decision making is uninterpretable. But we can look exactly at how our initial model is functioning, and see what features and cutoff values are important:

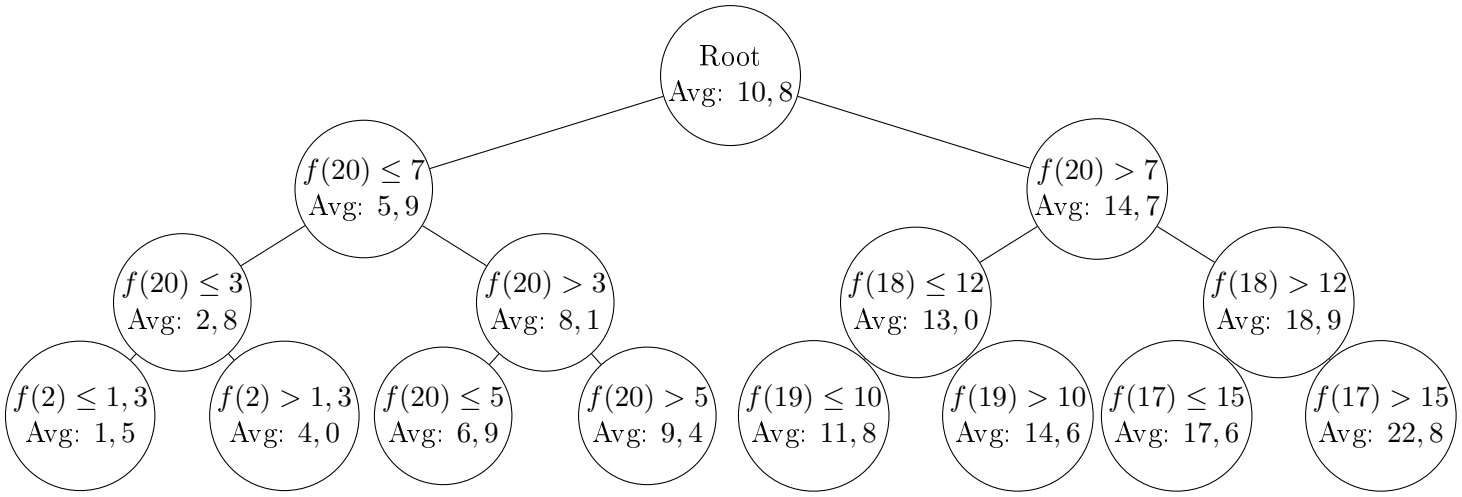


Figure 7: The first four layers of the decision tree model. In this context, $f(x)$ stand for the selected feature:

- $f(20)$: RegionNPSprctles[9]
- $f(19)$: RegionNPSprctles[8]
- $f(18)$: RegionNPSprctles[7]
- $f(17)$: RegionNPSprctles[6]
- $f(2)$: AvgNPS

The top and average densities seem to be the most important defining features for the initial layers. Note that splits can use the same features multiple times, as can be seen in one of the left paths. The model does pick up on other percentiles, as those provide information about how much of a chart is at higher note density.

3.3 Random Forest Regression

Random Forest Regression is a supervised learning method using ensemble learning for regression. Ensemble learning is a technique to combines predictions from several sub-models to make a more accurate prediction than a single model. Random Forest models also mitigate overfitting, as noise parameters are not likely to influence the outcome. This is very valuable to us, as initially we do not know which features are important, and so it is beneficial to be able to test as many as we can come up with, without negatively impacting performance. In

the regression variant of a random forest, we consider the output to be a continuous output variable even though the training output data only consists of integers. This isn't a problem, as we can view this difficulty level as a rounded integer of the actual underlying continuous spectrum.

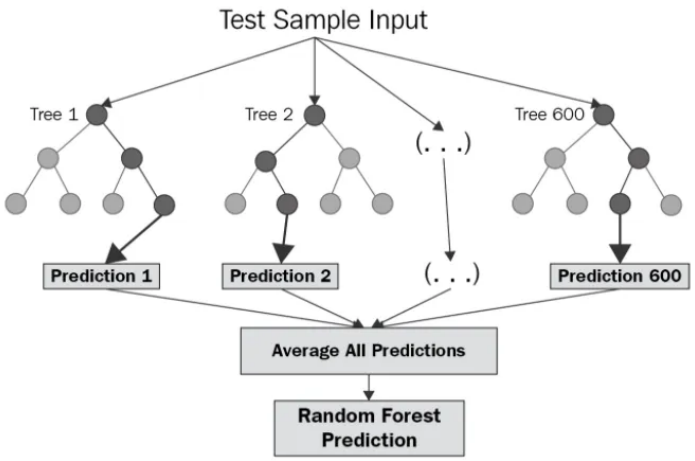


Figure 8: Diagram of the general structure of a Random Forest.[7]

For our model, we use a Python model and the scikit-learnpackage[8] to build the model. As it's a regression model, we expect important features contribute positively or negatively to the difficulty prediction, and want to measure the importance and impact of these features in the initial model to be able to fine-tune it in a later stage.

3.3.1 Initial Model

For the initial forest, we use the default set of 29 features as defined in the general methodology section. We build our initial model with the following specifications:

Table 4: Initial Model Parameters

Parameter	Value	Clarification
n_estimators	100	Number of trees in the forest.
criterion	"squared_error"	Mean squared error. Minimizes the L2 loss using the mean of each terminal node.
max_depth	None	Max depth of tree. "None" means we expand until all leaves are pure or contain less than the specified minimum samples.
min_samples_split	15	Minimum required samples before splitting a node.
min_samples_leaf	2	Minimum samples required to be at a leaf node.
max_features	None	Number of features to consider when looking for best split.Setting this to "none" means we use a maximum of all features.
oob_Score	True	Whether to use out-of-bag samples to estimate generalization score.

We then built our random forest regression model in Python, using the RandomForestRegressor class from the sklearn.ensemble module. The training data is a randomized selection of

[7] Source: https://levelup.gitconnected.com/random-forest-regression-209c0f354c84
[8] https://scikit-learn.org/stable/

90% of our database, leaving 10% for testing purposes.[9]

3.3.2 Initial Results

One advantage of a random forest regression model is that, unlike a lot of machine learning models, it isn't as much of a black-box. There are several tests and metrics to inspect the workings of the model. But let us first inspect the model's general performance:

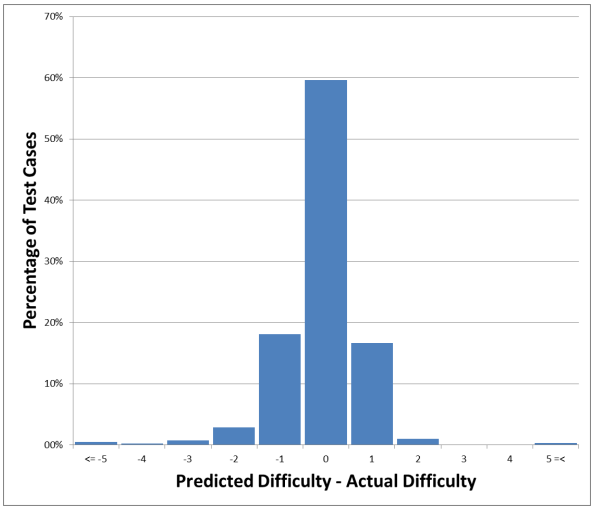


Figure 9: Histogram depicting how accurately the model is able to predict the difficulty ratings.

The buckets are defined with an interval of ± 0.5 centered around the test rating. So for example, a test rating of 14 is considered to be predicted accurately if the model outcome is between 13.5 and 14.5. Note that being 1-off might not be an inaccuracy of the model, but an indication of an initial mis-rating of the chart. After all, the purpose of these models is to aid in the subjective process of rating charts and identify mis-rated charts. Considering that, a total of 94.3% is identified correctly within a margin of 1.
Next, let us look at the contribution of each feature to the predictive capability of the model:[10]

[9] For the code, see main/Random Forest Regression/RandForest Build.py in the git project.
[10] For the code, see main/Random Forest Regression/RandForest Test.py in the git project.

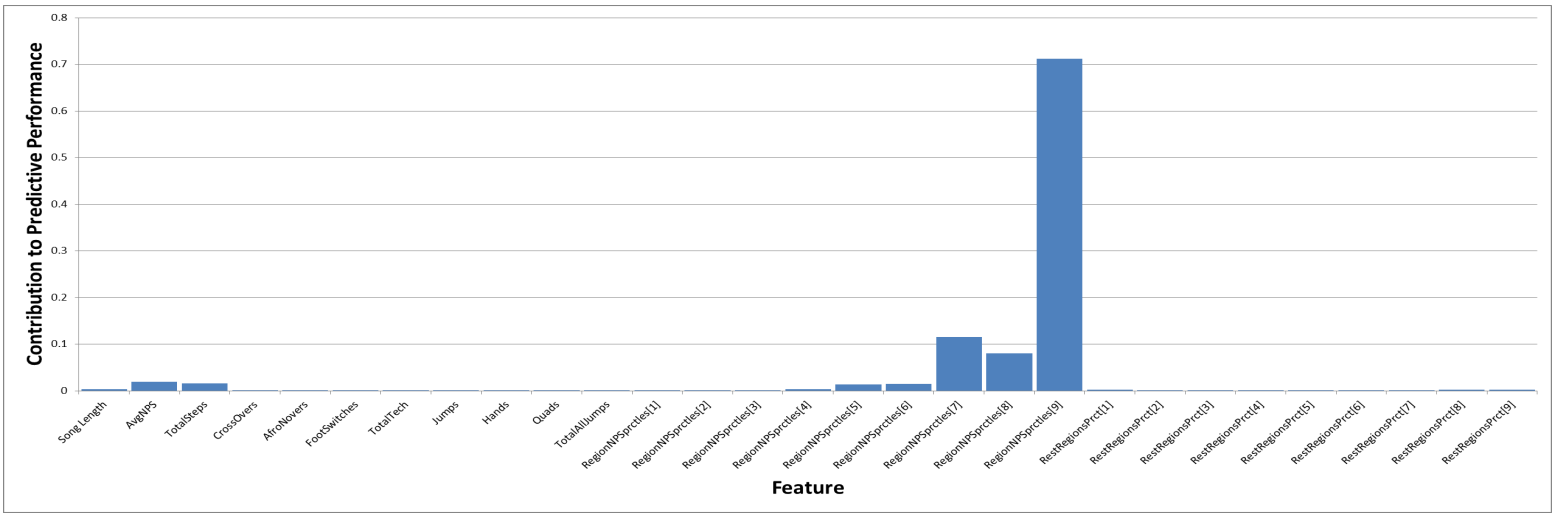
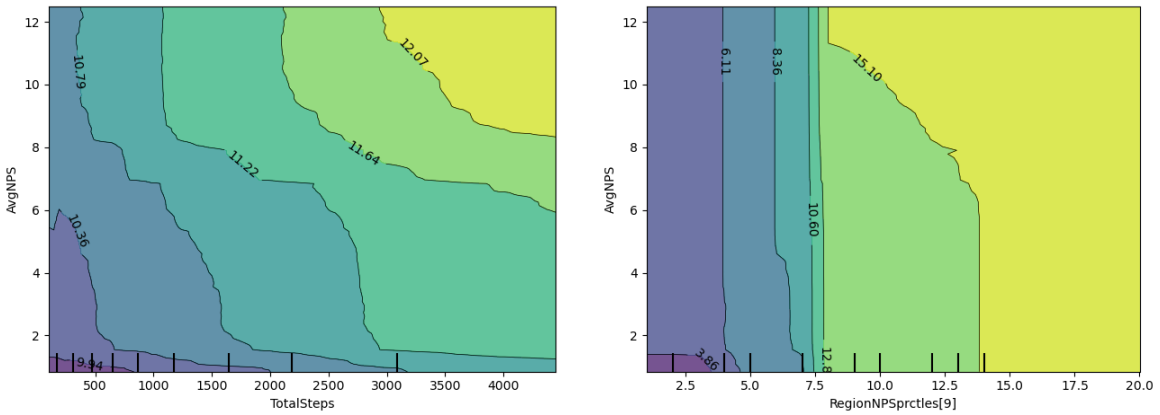


Figure 10: Feature importance of the initial random forest regression model.

It is immediately clear that almost all weight is centered around the fastest part of a chart. This is somewhat intuitive, as the quicker you'd have to move, the more difficult a chart. But this might not capture charts like marathons very well, which have their difficulty attributed to being very long and consistent, rather than having a high peak intensity. The model might also struggle with technical charts, which are rated higher due to their patterns. We will investigate this further in the fine-tuned model, where we look at some specific charts and see how the model performs with them.
As machine learning models can quickly become a black box of input and output, it is a good-practice to do some sanity checks. One way to do this, is by looking at partial dependence plots. Here we can see if multiple features interact with each other in an intuitive manner.



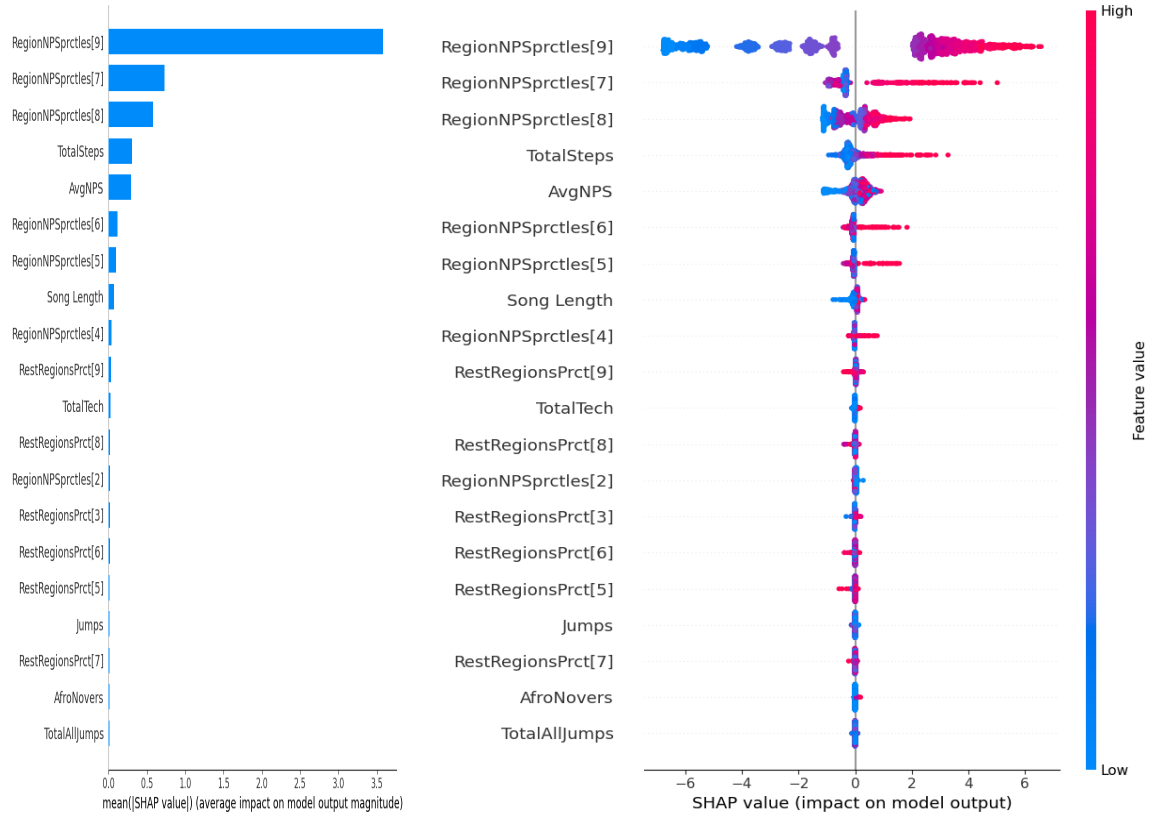
(a) Average steps/second and the total steps.

(b) Average steps/second and 10% fastest region.

Figure 11: Partial dependence contour plots showcasing dependency between two features. The vertical tick-marks on the x-axis indicate decile cutoffs.

Figure 11(a) paints a very intuitive picture where we see that both the more steps in a chart and the higher the average steps per second, the higher a rating will be. In Figure 4(b), we see that top-speed and average speed don't interact much in the lower top-speed regions. The average notes per second don't have as much of an impact until we get to the higher speed regions. There we see that for slower top-speed charts, a higher average density of steps is required to maintain the same difficulty. This makes sense as well - higher speed requires more breaks to be of the same difficulty.

Lastly we look at Shapley values, a method from coalitional game theory. This tells us how the contribution is distributed among the features.



(a) Bar graph.

(b) Scatter-plot.

Figure 12: Shapley values of initial random forest regression model.

It's surprising to see that the average impact on model output magnitude is only positive. We expect features like breaks to contribute negatively. In the scatter-plot, we do see that breaks contribute negatively in individual cases, but apparently on average the contribution is still positive.
We will later look at specific test-cases (select edge cases) and break down the Shapley values in those cases.

3.4 Random Forest Classifier

A Random Forest Classifier model is another example of a supervised learning method using ensemble learning for regression, like the random forest classifier. In the classifier variant of a random forest however, the output is a discrete class label. As it now no longer tries to reconstruct the output value directly out of the feature values but rather uses the feature values to classify them, this classifier model can pick up on features in a different way.

3.4.1 Initial Model

For the initial forest, we use the default set of 29 features as defined in the general methodology section. We build our initial model with the following specifications:

Table 5: Initial Model Parameters

Parameter	Value	Clarification
n_estimators	100	Number of trees in the forest.
criterion	"gini"	Using Gini to measure quality of split.
max_depth	None	Max depth of tree. "None" means we expand until all leaves are pure or contain less than the specified minimum samples.
min_samples_split	15	Minimum required samples before splitting a node.
min_samples_leaf	2	Minimum samples required to be at a leaf node.
max_features	None	Number of features to consider when looking for best split.Setting this to "none" means we use a maximum of all features.
oob_Score	True	Whether to use out-of-bag samples to estimate generalization score.

The initial model is built with these parameters.

We then again built the random forest regression mode in Python using the RandomForestRegressor class from the sklearn.ensemble module. The training data is a randomized selection of 90% of our database, leaving 10% for testing purposes.[11]

[11] For the code, see main/Random Forest Classifier/RandForest Build.py in the git project.

3.4.2 Initial Results

We again start by looking at the model's general performance:

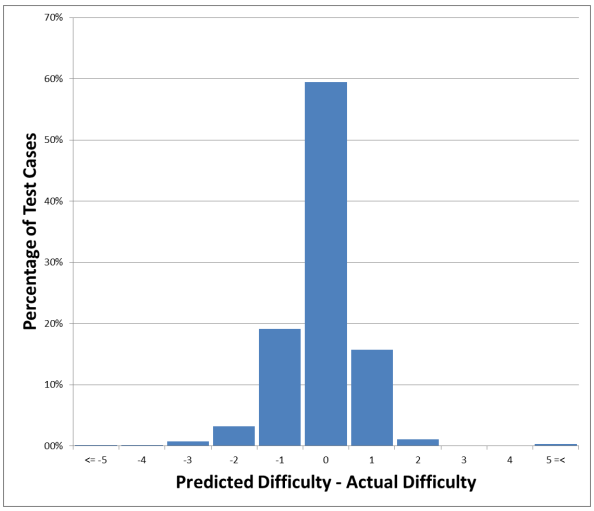


Figure 13: Histogram depicting how accurately the model is able to predict the difficulty ratings.

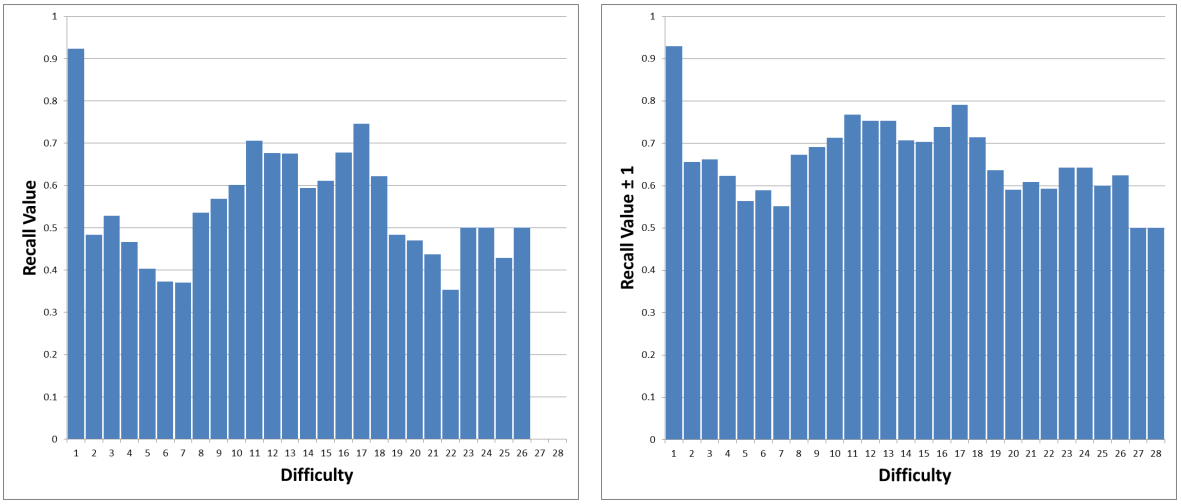
Similar to the random forest regression case, a total of 94.3% is identified correctly within a margin of 1. However, the distribution is slightly different and there seem to be fewer extreme outliers.

A metric exclusive to classifier models compared to regressors, is the recall value. Recall is the number of members of a class that the classifier identified correctly divided by the total number of members in that class.

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives}$$

In a regression model this value wouldn't make sense because regression involves predicting continuous values rather than classifying instances into discrete classes. In regression, predictions are a numerical value, and the evaluation metrics typically used are related to the accuracy of the predicted values compared to the actual values.[12]

[12]Something similar to recall could be achieved by bucketing predicted and true outputs, but better metrics exist for regression models.



(a) Recall value per target difficulty.

(b) Recall value within an error margin of ±1.

Figure 14: Recall values.

The mid-range difficulties and difficulty "1" seem to have the highest recall.[13] We notice that especially figure 14(a) bears a remarkable resemblance to figure 1, indicating that difficulties containing more training data, perform better in testing.
Now let's again look at the feature importance, where we'd expect the biggest difference as compared to the regression model.

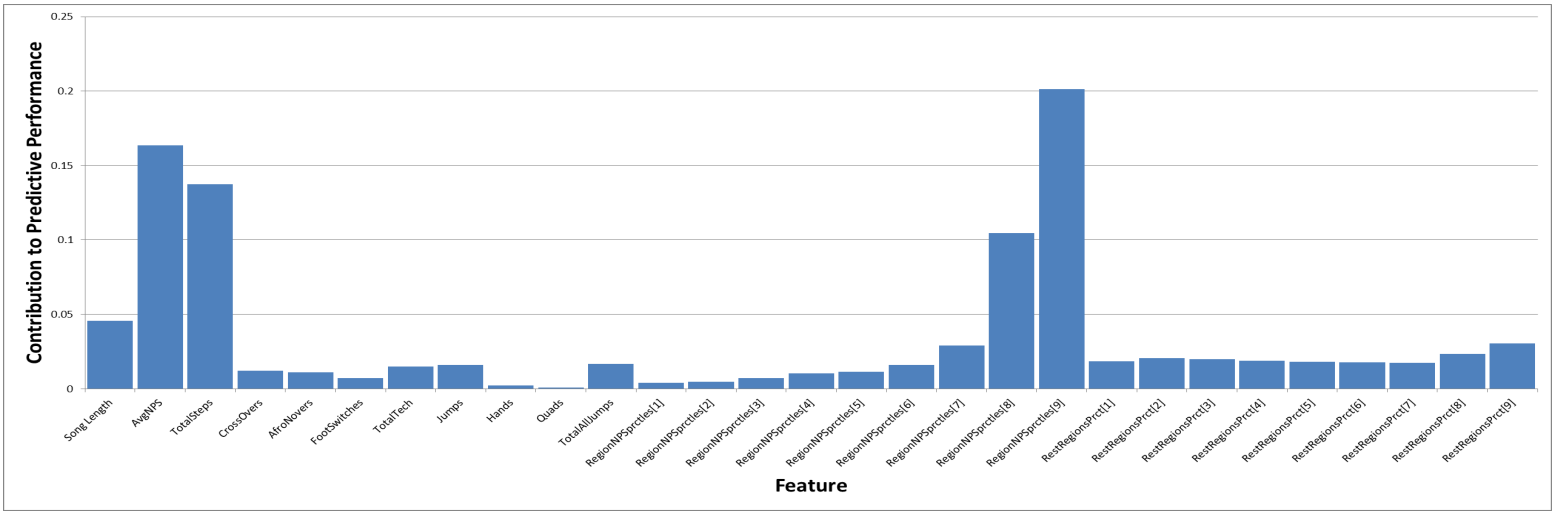
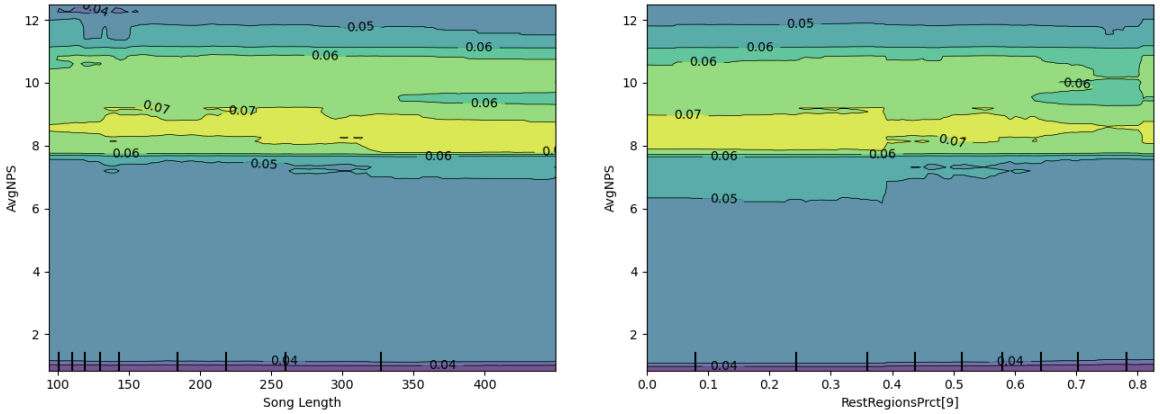


Figure 15: Feature importance of the initial random forest classifier model.

We can see that a lot more features are picked up with the classifier model as compared to the regression one. It recognizes that apart from the fastest regions, the average density and the total steps are important too. This indicates it picked up on the so-called "marathon" charts, which are longer songs with slower, but more consistent note density. This is strengthened by the fact that song length now also plays a bigger factor. Besides seemingly picking up on marathons, the model also recognizes that breaks are important, a feature almost completely disregarded by the regression counterpart. Lastly, the amount of technical patterns gain a

[13] For the code, see main/Random Forest Classifier/RandForest Test.py in the git project.

sleight increase in weight as well, although still not being a very prevalent feature.
To check if it indeed picked up on marathons and the importance of breaks, we create somesome partial dependence plots.



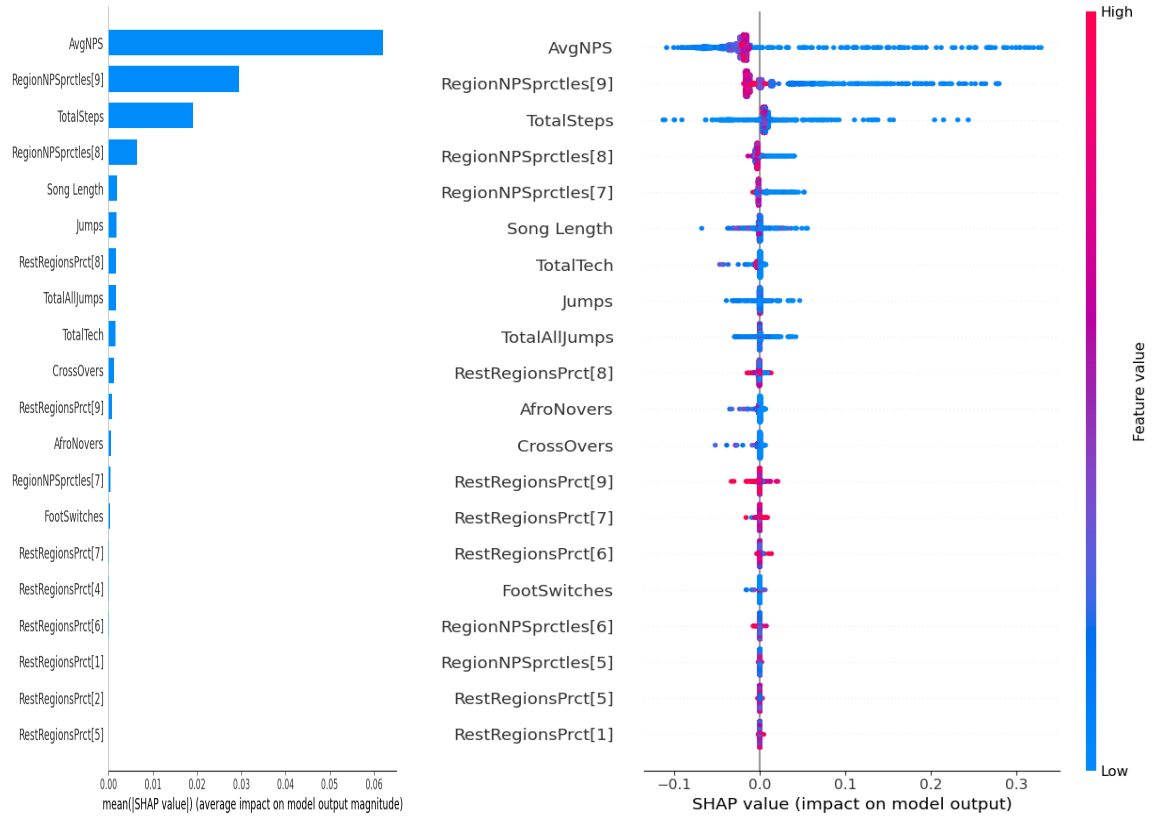
(a) Average steps/second and song lenth (category 15).

(b) Average steps/second and percentage rest regions. (category 15)

Figure 16: Partial dependence contour plots showcasing dependency between two features for a given category. The vertical tick-marks on the x-axis indicate decile cutoffs.

In figure 16(a), we expect to see the importance of the average steps per second to go up with song length, but that does not seem to be the case. We expect that there are just too few marathon charts, and there is not enough data for the x-axis to show longer songs to see what happens in the marathon region. We will later look at the model's behavior regarding marathon charts with the improved model. The same goes for the inclusion of breaks; we expect the importance to go up as the average density increases, but that's not clearly displayed in figure 16(b).

Lastly we look at the Shapley values again, to see if the contribution distribution among the features is significantly different from the regression random forest.



(a) Bar graph.

(b) Scatter-plot.

Figure 17: Shapley values of initial random forest classifier.

It is clear that the classifier model picked up on different and more features than the regression one. We first notice that now the greatest contributor seems to be the average density, instead of the peak density. It might look surprising that higher averages seem to contribute slightly negative to the difficulty when looking at figure 33(b). however, keep in mind that this is now a binary classifier model, and features just contribute to one side or the other. With that in mind, it clearly picks up that higher peak and average densities are important. We can also see that technical patterns now become a splitting feature, although the effect does not seem as strong. The same can be said about rests - on average, a high number of breaks seem to contribute well as a splitting feature.

3.5 Neural Network

A Neural Network is a machine learning model inspired by nature, it tries to emulate the interactions between neurons in a brain. It works by abstracting neurons to nodes which hold floating point numbers that linearly encode how active it is. Together, these nodes make up layers and the layers make up the whole network sequentially. For the purpose of simplification, the Neural Network model we used only has dense layers. This means that every neuron in a layer has a connection to every other neuron in the previous layer. Every connection between two neurons has a weight, which is a floating point number that linearly encodes how active the connection is.

3.5.1 Initial Model

We use three different types of densely connected layers in order:

- The input layer: this layer contains all of the normalized input data of a chart (its features). In this model we only use 27 of the 29 defined features, we left out TotalTech and TotalAllJumps as these are linearly dependent on already included features.
- Hidden layers: these are layers with no specified purpose, the goal of adding this is to pick up on certain patterns in the input data.
- The output layer: during testing we have used two different types of output layers:
  - Classification: each chart difficulty is assigned a neuron in the output layer. The floating point value between 0.0 and 1.0 attached to the neuron determines the probability ($\times 100\%$) that the neural network assigns to that answer being correct.
  - Regression: the output layer only holds one value between 0.0 and 1.0. The chart difficulty it predicts is determined linearly. So if the output layer holds x, where $x \in (0, 1)$, then the predicted difficulty ($D \in \mathbb{Z}$) is determined by:

    $$D = round(x \times D_{max})$$

    where D_{max} is the maximum difficulty of all charts, which is set to 38 in all tests.

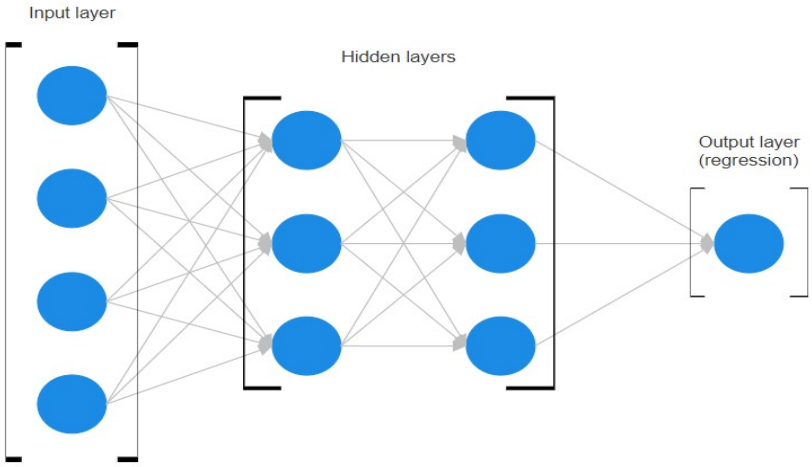


Figure 18: A graphical depiction of the model.

Four Activation functions have been tested. These functions are important for how information is passed between layers. Note that the *smht* function is just a modified version of the *tanh* function where we only choose one constant a for simplicity.

Table 6: Used activation functions.

Definition	Domain and Range
$sigmoid(x) = \frac{1}{1+e^{-x}}$	$\mathbb{R} \to (0, 1)$
$RELU(x) = max(x, 0)$	$\mathbb{R} \to (0, \text{inf})$
$tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	$\mathbb{R} \to (-1, 1)$
$smht(x) = \frac{ae^{x}-e^{-x}}{ae^{x}+e^{-x}}$, where $a \in (0, 1)$	$\mathbb{R} \to (-1, 1)$

We have used forward propagation (also known as forward pass) to pass data from its input to its output. This takes in two layers, the layer on the left (LHS) is used to calculate the neuron values for the layer on the right (RHS). The algorithm starts from the input layer and sequentially executes this for every layer except the output layer. Let $\vec{l} = (l_1, l_2, \ldots, l_n)^t$ and $\vec{r} = (r_1, r_2, \ldots, r_m)^t$ be vectors containing node values in the LHS and the RHS layers, where $n, m \in \mathbb{N}$ is the number of neurons in these layers respectively. Let $\vec{b} = (b_1, b_2, \ldots, b_m)^t$ be the vector of biases corresponding to $\vec{r}$. Let W be a $m \times n$ matrix of weights, where the rows are vectors corresponding to $\vec{l}$. Let f_a be an activation function with domain $\mathbb{R}$. Then:

$$r_i = f_a((\sum_{j=1}^{n} W_{ij} * l_j) + b_i)$$

The complexity of this algorithm can be expressed as $O(mn)$.

Next, we have to define a cost function to calculate the error (or loss) of the neural network's output. Note that this function takes in all input layer values, weights and biases because the output layer itself is calculated using all of these values. This is left out in the following equation for convenience. The way we calculate the cost is with mean squared error. Let $\vec{o} = (o_1, \ldots, o_n)^t, \vec{o} \in (0, 1)^n$ be the vector containing the output values of the Neural network after execution of forward propagation. Then let $\vec{e} = (e_1, \ldots, e_n)^t, \vec{e} \in (0, 1)^n$ be the vector containing the correct output values. During testing, $n = 1$ for regression type output and $n = 38$ for classification type output layer. Now we have:

$$Cost(\ldots) = \frac{(o_1(\ldots) - e_1)^2 + \ldots + (o_n(\ldots) - e_n)^2}{n}$$

The complexity of this algorithm can be expressed as $O(n)$. The goal of the neural network will be reducing the output of this cost function much like the previously mentioned models.

Backwards propagation is similar to forward propagation, this algorithm takes in two layers (LHS and RHS), but as the name suggests, it decides LHS by values in RHS. The way this is done can be understood by the process of gradient descent. We can see the entire neural network as a function that takes in the input layer values, weights and biases and returns the predicted difficulty of the chart. Every cycle of forward and backwards propagation updates all of the inputs of the entire neural network. These variables are updated according to the partial derivative of that variable in respects to the cost function. This means, for example, that if a variable increases the cost function, then backwards propagation will ensure it has less influence the next cycle. When a neural network has been sufficiently trained, the output of the cost function will converge on a local minimum. Think of it as a ball rolling down a 3-dimensional landscape and increasingly slowly approaching a valley. In reality, this landscape is n-dimensional, where n is the number of variables of the entire neural network (input layer values, weights and biases) plus one (the output $\in \mathbb{R}$ which can be seen as the "height" of the ball). Similar to forward propagation, the complexity of this algorithm can be expressed by $O(mn)$, where n is the number of neurons in LHS and m is the number of neurons in RHS.

The model has a number of hyper-parameters we can set for testing purposes:

Table 7: List of hyper-parameters used during testing.

Hyper-parameter	**Description**
Learning rate	Determines how fast the neural network strides towards the local minimum during training. High learning rate values might destabilize the training process leading to ineffective training. Very low learning rate values might lead to the neural network taking a lot of training cycles before reaching a local minimum.
Activation function	See Table 6.
Epochs	The number of times the neural network will train every chart.
Hidden layer count	Number of hidden layers.
Neurons per hidden layer	This is an array that holds the amount of neurons for every hidden layer, indexed so that $[i]$ stores the amount for hidden layer $i + 1$.
Used inputs	Specifies which features are used as inputs. Some features may be counterproductive, and we want to be able to turn those off.

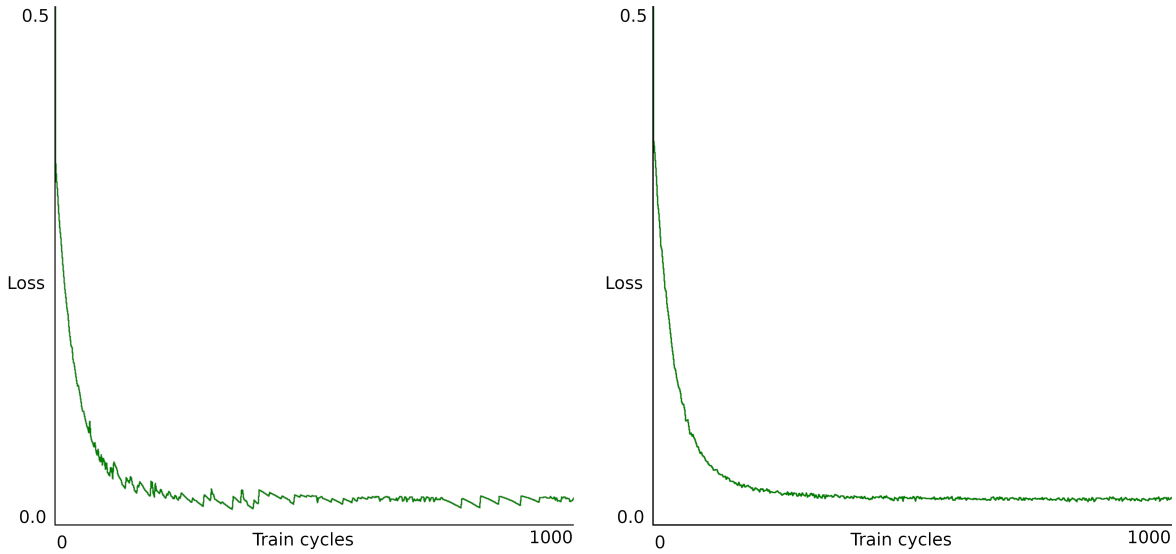
In the initial model, we used the following:

Table 8: Hyper-parameter settings for the initial model.

Hyperparameter	**Setting**
Learning rate	0.1
Activation function	Sigmoid
Epochs	1
Hidden layer count	1
Neurons per hidden layer	128
Used inputs	All $(0 - 27)$

Additionally, the used output layer is a classification type output layer.

3.5.2 Initial Results



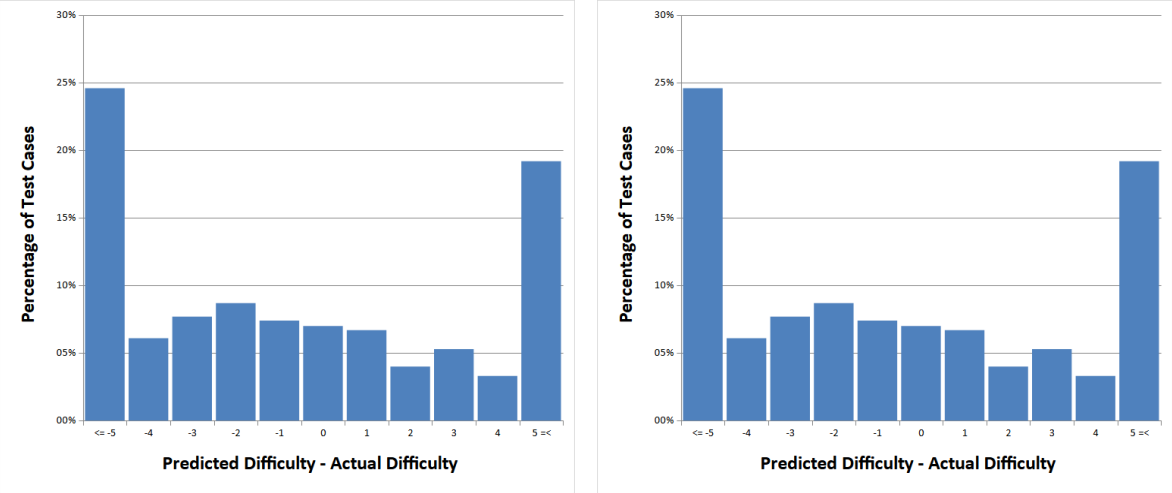
(a) Before adding randomization.

(b) After adding randomization.

Figure 19: Training loss graphs.

If we look at figure 19 (a), something very peculiar is happening. The loss during training seems to consistently decrease at certain intervals before shooting back up. This was caused by the fact that the charts were not being randomized. There is a correlation with the name of the chart and the difficulty of the chart, causing the over-fitting. The loss graph on the right is from after adding the randomization to all the charts. The decrease in loss is now smooth where it previously was bumpy.
Even after correction, the results are still mostly noise. We suspect that the neural network was doing informed guesses. To test this, we compare the results of two runs. One of which all features are turned off, and the other one where every feature is turned on (see figure 20). The results are exactly the same, meaning the neural network was not using the features to train at all.[14]

[14] All raw results and graphs of the tests can be found in main/NN/res/inputTests of the git project.



(a) All features enabled.

(b) No features enabled.

Figure 20: Histogram depicting the prediction accuracy of the classification model input tests.

In order to overcome this unwanted behavior, we require a new way to punish and reward the network. As mentioned, at this point the neural network is still using a classification system to predict difficulties of charts. The problem with this approach is that difficulty is continuous, that is: if the actual difficulty of a chart is x, then a predicted difficulty of $x+1$ is more correct than predicted difficulty of $x+2$. This being "close enough" might not be getting rewarded enough in the classification model as it only considers one answer to be correct. We look into whether the regression model does better in the improved model section.

4 Initial Model Insights for Improvements

In the initial results section of the models, we've had several observation and noticed improvements to be made to each model. First and foremost, the initial parameters on which the models were built can be adapted and fine-tuned for better results. In the case of the linear regression, we can try to shift some of the model weight away from just the average steps per second to make it more sensitive to other explanatory variables. The random forest regression model had similar issues. The decision tree seemed insensitive to especially the pattern related features. Breaks were also not picked up on much by most models, bare perhaps the random forest classifier.

To improve the models, we start by introducing two new features, and changing the way the "RestRegionsPrct" features are calculated. Initially, breaks at the start and end of each chart were included in the percentage of the chart that was considered a rest. In practice, these parts can be disregarded, and should have no influence on the difficulty. Therefore, we re-defined this feature to count only from after the first breach of an intensity threshold, up to the last one.

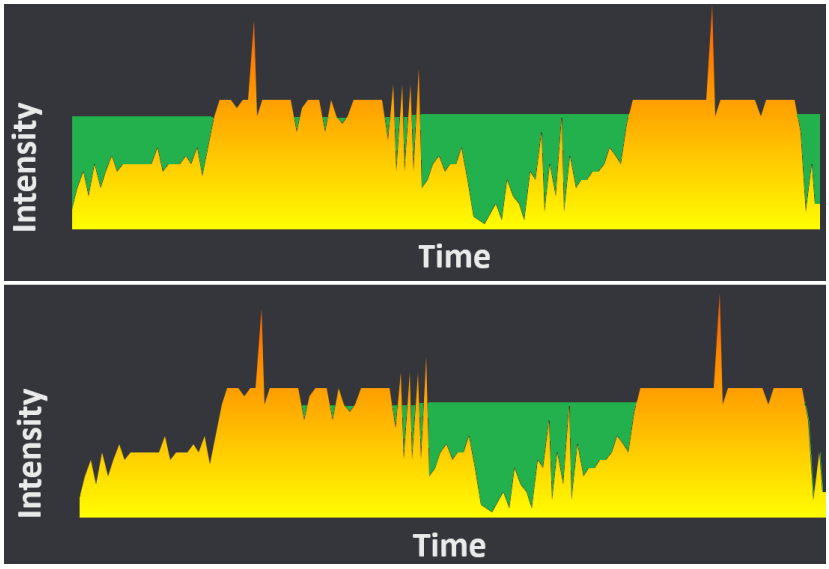


Figure 21: The top graph shows in green what was considered "rest" before the change, the bottom graph after the change.

The first new feature we call "TotalPatterns" and is simply the sum of "TotalAllJumps" and "TotalTech", in an attempt to give more weight to more technical patterns. Spreading a feature out over too many attributed can lead to models like decision trees or random forests to not select these variables, as their individual contribution might not stand out enough. By combining these into a single variable, we attempt to give more weight to it.
The second new feature is called "Intensity", which is a single number that increases as the chart density is above $0.75 \times RegionNPSprctles[9]$ and decreses below. [15]
Finally, we will test some specific pairs of charts to see if the models can distinguish them based on specific attribute. These charts are identical to each other, with the exception of a single dimension.

Table 9: Specific Test Charts

Chart Label	Difficulty	Characteristics
OL	14	Marathon chart of 15 minutes long, with constant density and no technical patterns.
WAO	17	Feature wise nearly identical to OL, except the chart is 1 hour long.
GB13	13	Regular chart with no technical patterns.
GB15	15	Identical in every way to GB13, except the patterning is more technical.
EE	16	Relatively fast for the difficulty, but in shorter sprints.
SV	18	Similar to OS, but faster.

Specific charts to test if models can pick up on specific feature differences.

[15] See main/AV Beta/scripts/Parse-DB-RegTree/main.lua in the git project for code including the precise calculation.

OL and WAO will be used to test if models picked up on how length contributes to difficulty. These two charts are identical, with the exception that WAO is four times longer. Likewise, GB13 and GB15 are completely identical, bare the patterning. That way, we can test which models picked up on how stepping patterns contribute to difficulty. Lastly, OS and US are relatively similar, with short sprints and many breaks in between, the difference being the speed of the sprints. We will look at the predictions for these charts with each model, as a way to test the predictive capabilities based on specific feature differences.

5 Improved Models

We will now attempt to better the models, using the results from the initial ones. In addition to the features from Table 1, we use two more variables:

Table 10: Initial Model Features

Name	Description	Expected Impact
Intensity	Abstract indicaation of difficulty, obtained by means similar to integrating over the density.	Higher Intensity correlates to a higher difficulty.
TotalPatterns	Sum of all technical patterns and all forms of jumps.	More complex stepping patterns increase the difficulty.

Features for the improved models, in addition to Table 1.

5.1 Standard Linear Regression

The initial model seemed to have too many parameters, and didn't pick up on the right weights for each feature. We will try to address this by working with a smaller model, consisting of only the most important features. Additionally, we removed variables of which linear combinations could lead to others. This is something to avoid in general when doing standard linear regressions.

5.1.1 Improved Model

After several tests and iterative model changes, the final selection of parameters for the standard linear regression model is:

Table 11: Improved Regression Model Features

Feature
Constant
Song Length
AvgNPS
TotalSteps
RegionNPSprctles[5]
RegionNPSprctles[6]
RegionNPSprctles[7]
RegionNPSprctles[8]
RegionNPSprctles[9]
RestRegionsPrct[7]
Intensity
TotalPatterns

Features selected for the improved linear regression model.

MatLab was used to build this linear regression model. The training data is again a randomized selection of 90% of the database with the remaining 10% used for testing. Linear regression is then performed on the training data, with heteroskedasticity and autocorrelation corrected standard errors.

5.1.2 Improved Model Results

After recalculating the β vector from the model, we test the performance:

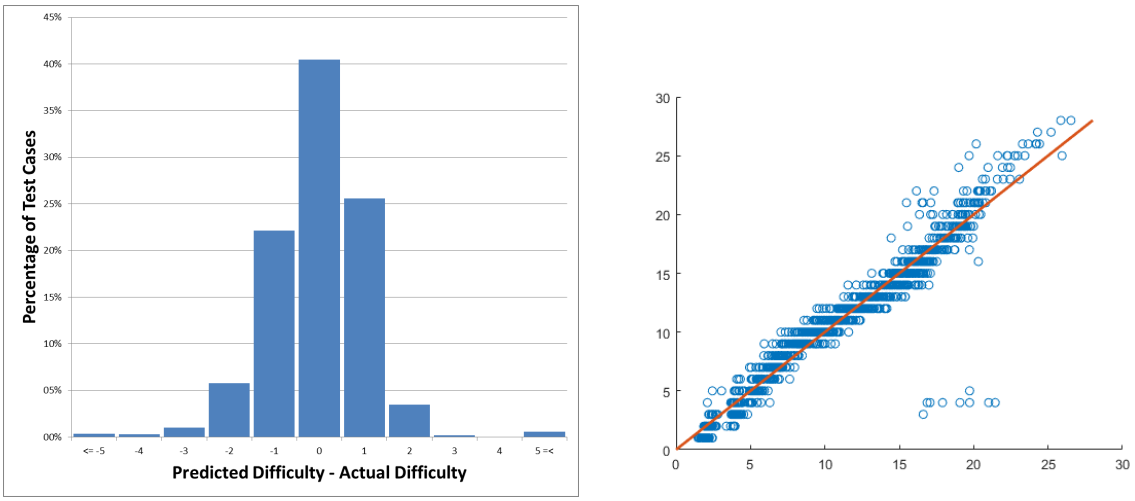


Figure 22: Histogram depicting the prediction accuracy of the improved linear regression model.

The buckets are defined with an interval of ± 0.5 centered around the test rating. In total, 88.2% is identified correctly within a margin of 1, with 40.5% matching perfectly. This seems worse than the initial model, but we are using far less parameters this time. The cluster in the bottom center of the scatter plot is still present, though some other outliers seemed to have cleaned up. With linear regression, it's possible to make this metric arbitrary large by introducing an arbitrary amount of variables, resulting in over-fitting. Considering we are only using 11 features instead of 29, the drop in performance is negligible.
Next, let's look again at the meta-data regarding the performance of the model:

Table 12: Meta statistics about the initial linear regression model.

Statistic	Value
R^2	0.93
F-Stat	15770
P-value	0
estimate of error variance	2.34

The R^2 remains unchanged, despite having so much fewer explanatory variables. Unfortunately, when looking at the weights in the β array it is still clear that the model isn't picking up on the features the way we would expect it to.

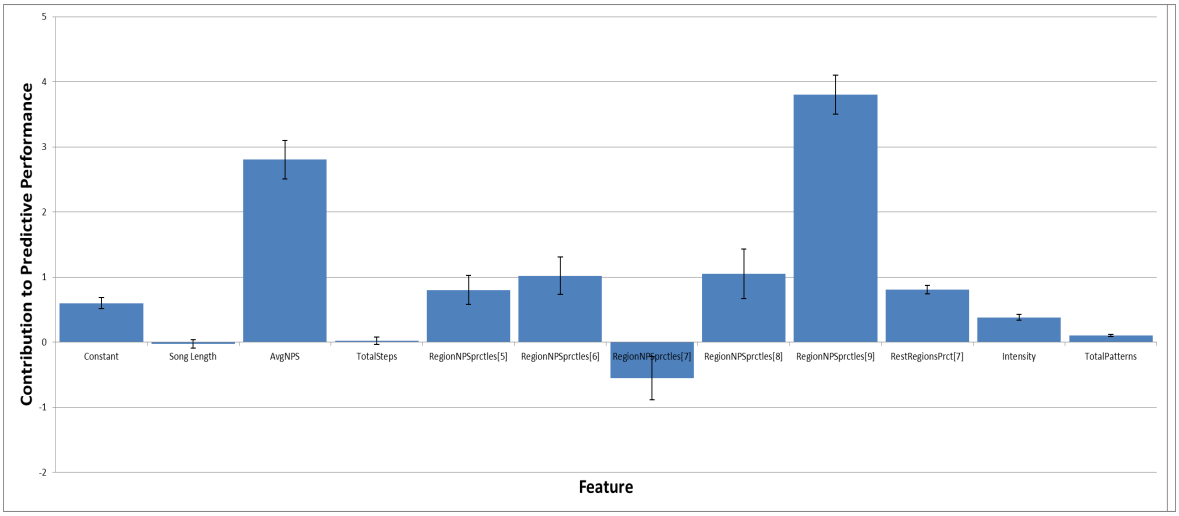


Figure 23: Normalized feature weights (β) of the initial linear regression model, within 95% certainty.

The models focus has shifted from the average density, to a combination of the average and top densities. It is expected that the faster percentile regions would contribute positive, and for the rest regions to contribute negatively. And in a way this happens, although the 70% percentile fastest region contributes negatively, and the rest region positively. If these two were swapped, the model would behave exactly as expected. The 95% confidence intervals also seem to be less wide in general.

Finally, we look at the predictions for our set of test charts.

Table 13: Predictions on specific test charts.

Chart Label	Actual Difficulty	Predicted Difficulty
GB13	13	13.87
GB15	15	14.33
OL	14	14.88
WAO	17	18.74
EE	16	16.94
SV	18	18.30

We do see that all attributes get accounted for. Most charts are predicted accurately as well (note that the actual difficulty is similar to a "floor" function; a 13.9 would still be considered "13"). It estimates GB15 to be about half a difficulty higher than GB13 based on patterning alone. It does overestimate the impact of length on WAO a lot, and both EE and SV are predicted accurately. Length and patterns seem to be the hardest features to accurately predict the impact off, as expected by the model analysis.

5.2 Decision Tree

The initial model performed acceptable already, but we'll try to improve it by tuning the dimensions of the tree, the minimum number of observations in each leaf and the granularity of checking for cutoffs.

5.2.1 Improved Model

We use the same explanatory variables as described in Table 1, with the addition of the aforementioned "intensity" and "TotalPatterns" features and the new definition of rest regions. Using MatLab to build the decision tree, we arrive at two configurations: one for the most exact accurate predictions, and one for the best performance considering predictions within 1 off:

Table 14: Initial Model Parameters

Parameter	Value Exact	Value 1-Off	Clarification
MaxTreeLength	12	15	Maximum depth of the tree.
NSteps	1000	10000	The granularity of which feature values are picked for splitting: $\frac{Max\_ft.val - Min\_ft.val}{NSteps}$.
MinLeaves	9	9	Branch won't be split if the number of samples $\leq$ MinLeaves.

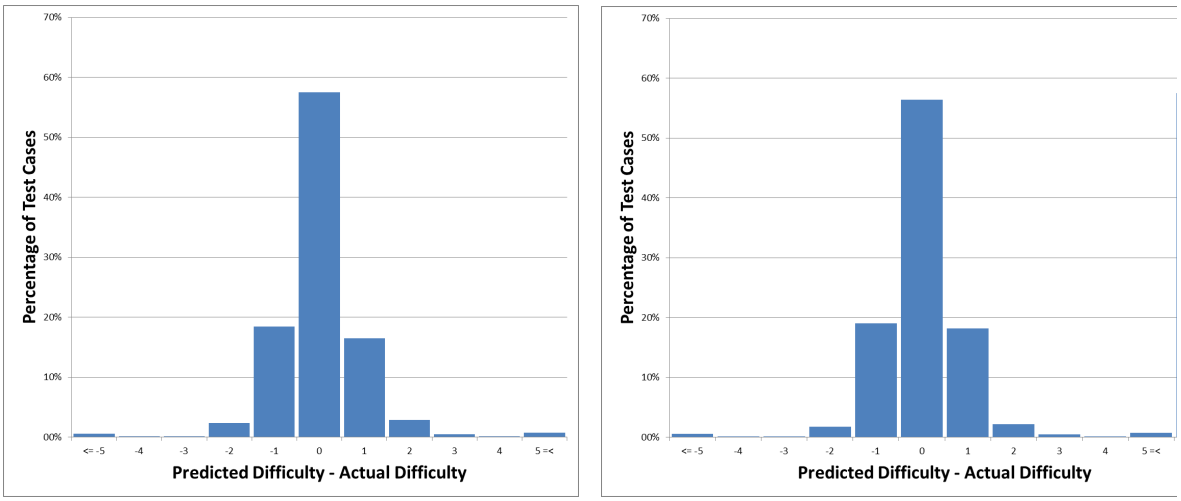
The initial decision tree is built with these parameters.

Training data is again a randomized selection of 90% of the database with 10% for testing purposes.[16]

[16] For the code, see main/Decision Tree/Plant_DesTree.m in the git project.

5.2.2 Improved Model Results

Running the test set on the improved model, we arrive at the following results:



(a) Calibrated for highest exact match.

(b) Calibrated for accuracy within 1-off.

Figure 24: Histogram depicting the prediction accuracy of the improved decision tree model.

The buckets are defined with an interval of ± 0.5 centered around the test rating. For the exact-match model, 92.4% is predicted correctly within 1-off, and 57.5% of the test cases were predicted with an exact match. For the 1-off model, the numbers are 93.6% and 56.4% respectively. Fine tuning resulted in a gain of 7.2% in 0-off accuracy, and 2.6% in 1-off accuracy. This accounts for an about 30% increase in performance (from 8.9% mis-rated charts down to 6.4%).
When looking at the first couple of layers in the tree, these seem mostly unchanged compared to the initial model, and are identical for the two optimized models. This is to be expected however, as the first couple of splits are the most coarse and done with the most impactful features. These are unlikely to change when fine-tuning.

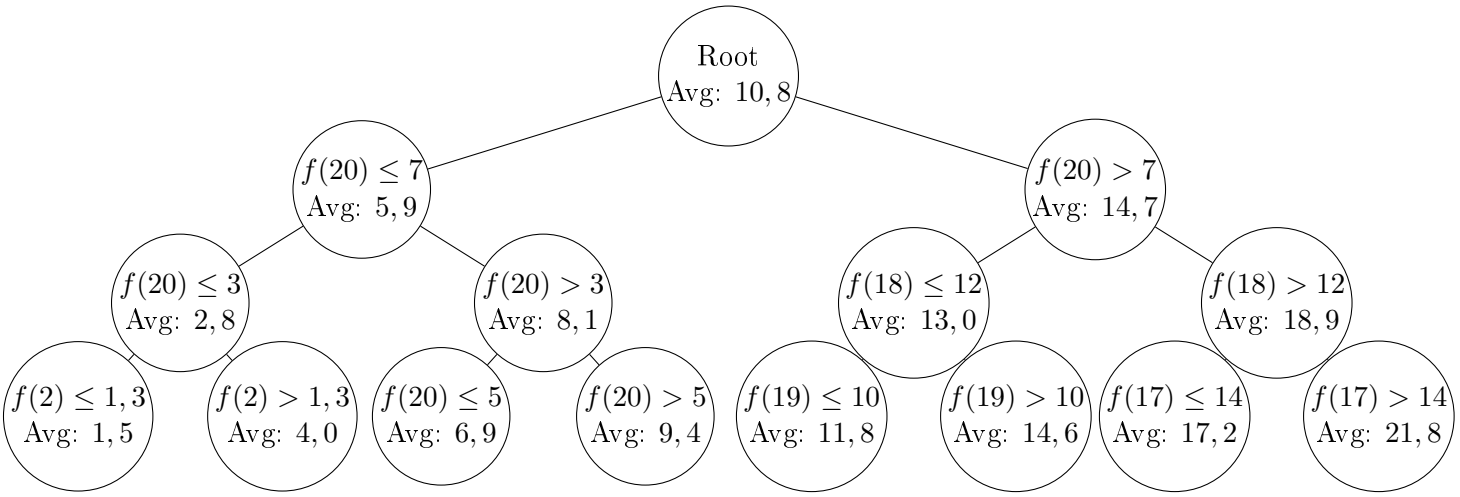


Figure 25: The first four layers of the decision tree model. In this context, $f(x)$ stands for the selected feature:

- $f(20)$: RegionNPSprctles[9]
- $f(19)$: RegionNPSprctles[8]
- $f(18)$: RegionNPSprctles[7]
- $f(17)$: RegionNPSprctles[6]
- $f(2)$: AvgNPS

When inspecting further layers of the tree however, we notice that the newly added intensity and total_patterns features start making an impact starting from layer 4 and 6 respectively. Especially the intensity feature is used often in the middle layers of the tree, with total_patterns often being used in the bottom three layers. This is good, as it recognizes that two otherwise very similar charts can be distinguished by their patterning.
Next we take a look at the performance with the set of specific test charts:

Table 15: Predictions on specific test charts.

Chart Label	**Actual Difficulty**	**Prediction 0-Off Model**	**Prediction 1-Off Model**
GB13	13	13.96	14.00
GB15	15	13.96	14.00
OL	14	14.57	14.56
WAO	17	16.81	16.81
EE	16	16.83	16.83
SV	18	17.38	17.38

Both models are only accurate on three out of the six test charts, and are not capable of distinguishing between GB13 and GB15, which only differ by patterning. It seems as if the decision tree performs very well on average, but breaks down when it comes to niche cases. In order to counteract this, we can use a much deeper tree with a lower parameter for minimum leaves. If for example we use a tree with the following parameters:

Parameter	Value
MaxTreeLength	21
NSteps	1000
MinLeaves	2

then we obtain the following predictions on the test charts:

Chart Label	Actual Difficulty	Prediction 0-Off Model
GB13	13	13.50
GB15	15	15.00
OL	14	15.00
WAO	17	16.50
EE	16	16.50
SV	18	13.17

At least now GB13 and GB15 are distinguished between, though an argument could be made that that is the result of over-fitting. We can check this by looking at the number of samples in the corresponding leaves. For GB13, there are 7 samples in the final leave and for GB15 there are 2. This does not seem to be over-fitted for GB13 at least. A final observation is the way off mis-prediction for SV. This is typical for a decision tree, as where regression models try to compute the output from explanatory variables, decision trees just categorize samples based on similarity. The average of the three samples of the leaf SV is in, is 13.17. This might suggest a "waste leaf", where odd or hard to classify samples were lumped in to. Adding more layers could resolve that, though is prone to over-fitting. Further improvements could be made by doing meta-analysis, such as looking at the number of samples per leaf and analyzing the peer groups in each leaf to see if indeed similar charts end up in the same leaf, or that modifications might be necessary.

5.3 Random Forest Regression

In order to improve the random forest regression model, we fine-tune the model parameters line the number of trees, feature selection algorithm and the number of features to consider when looking for best split.

5.3.1 Improved Model

For the improved forest, we use the default set of 29 features as defined in the general methodology section, with the modified rest region calculations and the intensity and total pattern features added. Using the sklearn Python package, after trying many different configurations and doing impact analysis on the parameters we arrive at an improved model with the following specifications:

Table 16: Improved Model Parameters

Parameter	Value	Clarification
n_estimators	225	Number of trees in the forest.
criterion	"squared_error"	Mean squared error. Minimizes the L2 loss using the mean of each terminal node.
max_depth	None	Max depth of tree. "None" means we expand until all leaves are pure or contain less than the specified minimum samples.
min_samples_split	6	Minimum required samples before splitting a node.
min_samples_leaf	3	Minimum samples required to be at a leaf node.
max_features	None	Number of features to consider when looking for best split.Setting this to "none" means we use a maximum of all features.
oob_Score	True	Whether to use out-of-bag samples to estimate generalization score.

With this, the random forest regression model is built in Python, using the RandomForestRegressor class from the sklearn.ensemble module. Once again training data is a randomized selection of 90% of our database, leaving 10% for testing purposes.[17]

5.3.2 Improved Model Results

We first inspect the model's general performance, before taking another look at the meta metrics, and how the model has changed.

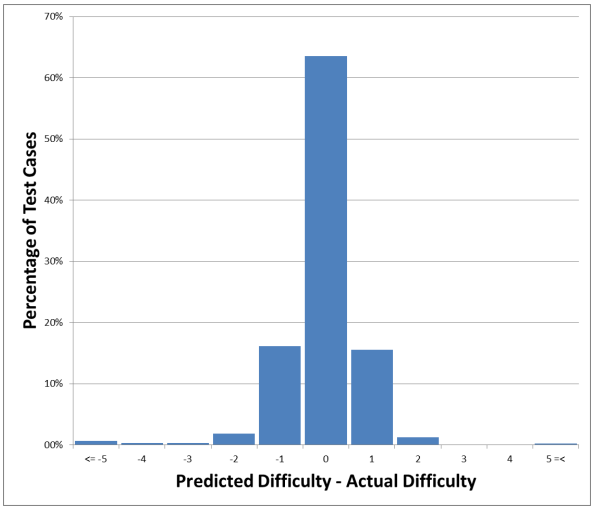


Figure 26: Frequency of accurate predictions with given error.

The buckets are defined with an interval of ±0.5 centered around the test rating. A total of 95.2% is identified correctly within a margin of 1. This is a 0.9% point improvement to the initial model, or a relative increase of 16% in performance (from 5.7% mis-rated charts down to 4.8%).

[17] For the code, see main/Random Forest Regression/RandForest Build.py in the git project.

Now, let's look again at the contribution of each feature to the predictive capability of the model:[18]

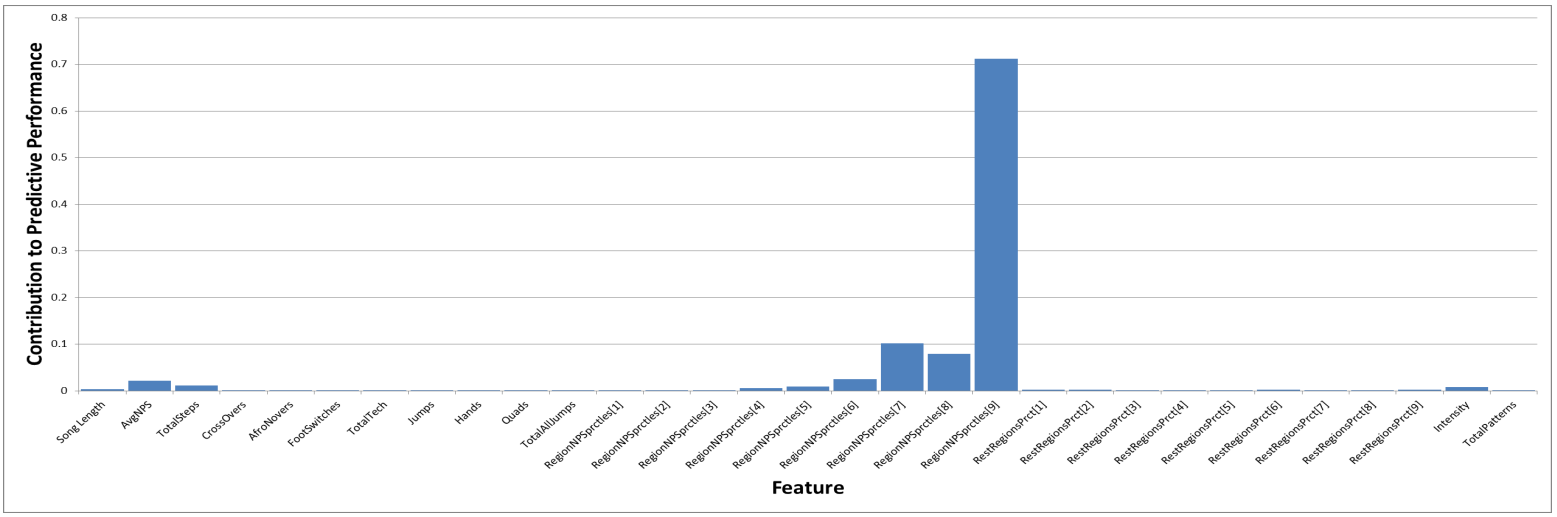
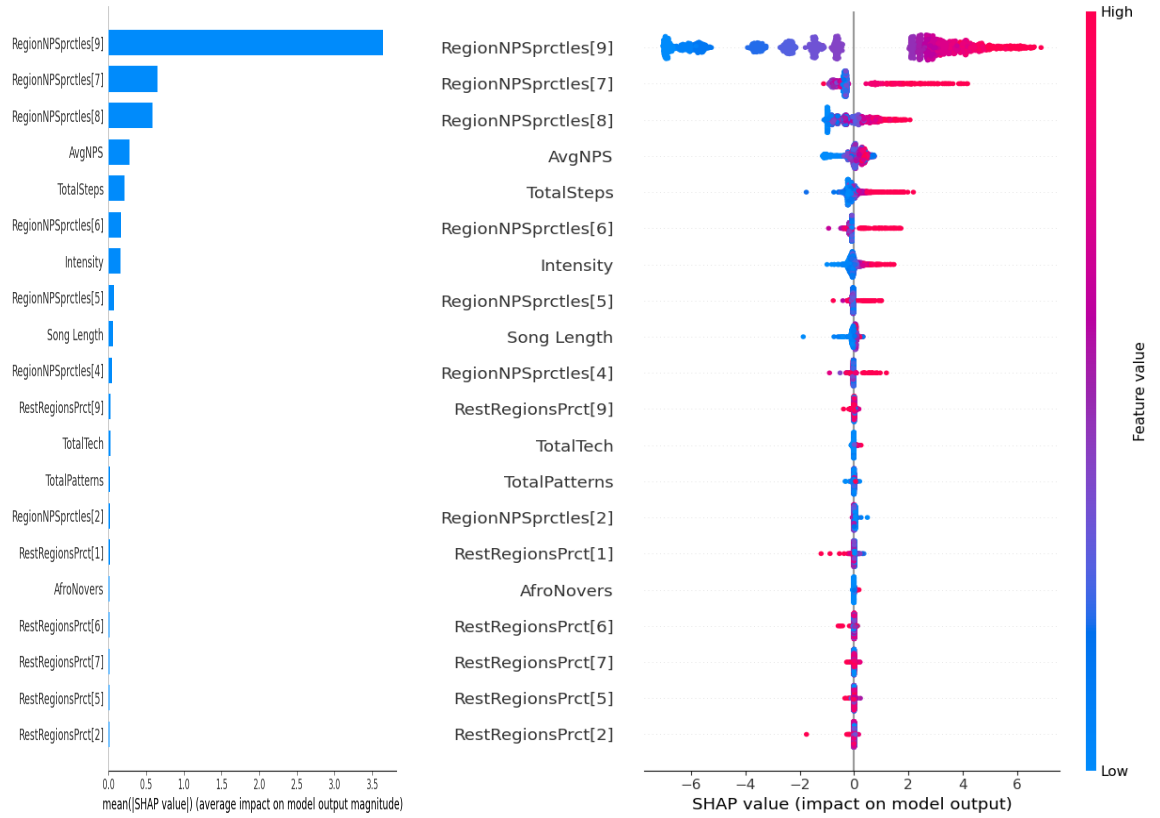


Figure 27: Feature importance of the improved random forest regression model.

The feature importance is very similar to that of the initial model, and the intensity feature doesn't get picked up on as much as we had hoped. It still is about the 7th most important contributor to the model, on par with average density. We'd like to optimize this feature further by changing the algorithm when parsing charts, but that is left as a future improvement.

[18]For the code, see main/Random Forest Regression/RandForest Test.py in the git project.

Let's now revisit the Shapley values, and look at how the contribution is distributed among the features for the improved model:



(a) Bar graph.

(b) Scatter-plot.

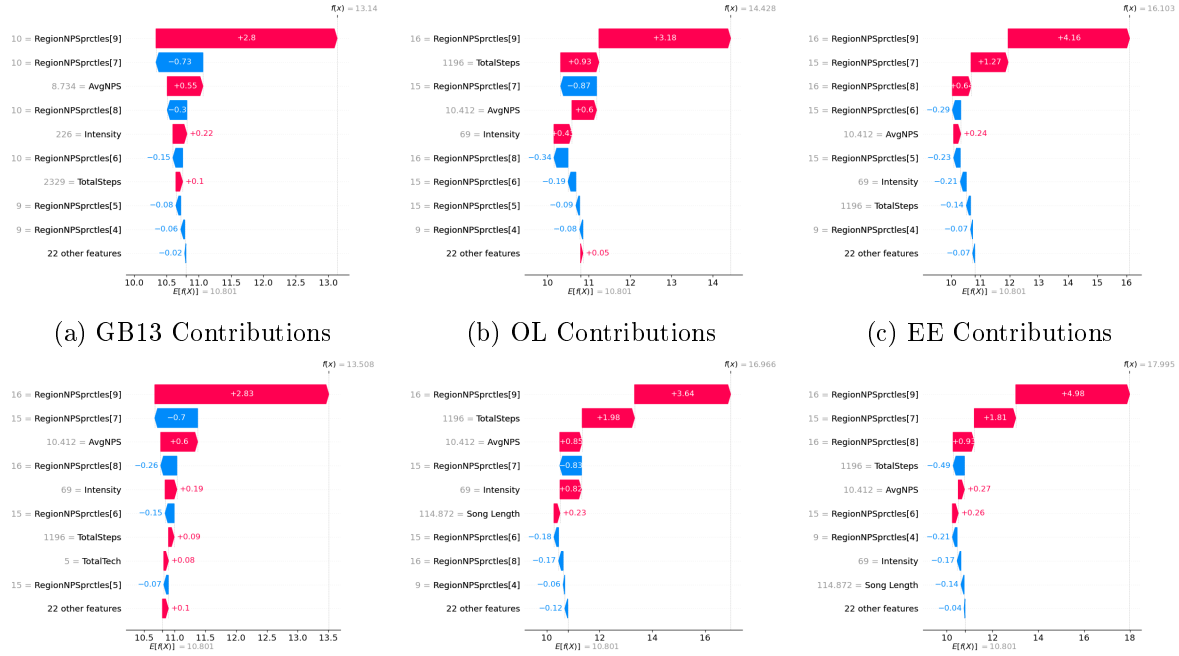
Figure 28: Shapley values of initial random forest regression model.

There are no major changes as compared to the initial model, except some slight shifts in a few variables.
Other than looking at Shapley values for the entire population, we can also look at the values for individual charts. We will look at this using our test charts.

Table 17: Random forest regression predictions on specific test charts.

Chart Label	Actual Difficulty	Prediction Random Forest Regression
GB13	13	13.14
GB15	15	13.51
OL	14	14.43
WAO	17	16.97
EE	16	16.10
SV	18	17.99

The accuracy seems impressive, though it has a hard time picking up on patterns, even with the new "TotalPatterns" feature. We can inspect this further by looking at chart-specific Shapley value breakdown, to see which elements contribute most for each chart.


f(x) = 13.14
10 = RegionNPSprctles[9] +2.8
10 = RegionNPSprctles[7] −0.73
8.734 = AvgNPS +0.55
10 = RegionNPSprctles[8] −0.3
226 = Intensity +0.22
10 = RegionNPSprctles[6] −0.15
2329 = TotalSteps +0.1
9 = RegionNPSprctles[5] −0.08
9 = RegionNPSprctles[4] −0.06
22 other features −0.02
10.0 10.5 11.0 11.5 12.0 12.5 13.0
E[f(X)] = 10.801
f(x) = 14.428
16 = RegionNPSprctles[9] +3.18
1196 = TotalSteps +0.93
15 = RegionNPSprctles[7] −0.87
10.412 = AvgNPS +0.6
69 = Intensity +0.4
16 = RegionNPSprctles[8] −0.34
15 = RegionNPSprctles[6] −0.19
15 = RegionNPSprctles[5] −0.09
9 = RegionNPSprctles[4] −0.08
22 other features +0.05
10 11 12 13 14
E[f(X)] = 10.801
f(x) = 16.103
16 = RegionNPSprctles[9] +4.16
15 = RegionNPSprctles[7] +1.27
16 = RegionNPSprctles[8] +0.6
15 = RegionNPSprctles[6] −0.29
10.412 = AvgNPS +0.24
15 = RegionNPSprctles[5] −0.23
69 = Intensity −0.21
1196 = TotalSteps −0.14
9 = RegionNPSprctles[4] −0.07
22 other features −0.07
10 11 12 13 14 15 16
E[f(X)] = 10.801
f(x) = 13.508
16 = RegionNPSprctles[9] +2.83
15 = RegionNPSprctles[7] −0.7
10.412 = AvgNPS +0.6
16 = RegionNPSprctles[8] −0.26
69 = Intensity +0.19
15 = RegionNPSprctles[6] −0.15
1196 = TotalSteps +0.09
5 = TotalTech +0.08
15 = RegionNPSprctles[5] −0.07
22 other features +0.1
10.5 11.0 11.5 12.0 12.5 13.0 13.5
E[f(X)] = 10.801
f(x) = 16.966
16 = RegionNPSprctles[9] +3.64
1196 = TotalSteps +1.98
10.412 = AvgNPS +0.85
15 = RegionNPSprctles[7] −0.83
69 = Intensity +0.82
114.872 = Song Length +0.23
15 = RegionNPSprctles[6] −0.18
16 = RegionNPSprctles[8] −0.17
9 = RegionNPSprctles[4] −0.06
22 other features −0.12
10 12 14 16
E[f(X)] = 10.801
f(x) = 17.995
16 = RegionNPSprctles[9] +4.98
15 = RegionNPSprctles[7] +1.81
16 = RegionNPSprctles[8] +0.91
1196 = TotalSteps −0.49
10.412 = AvgNPS +0.27
15 = RegionNPSprctles[6] +0.26
9 = RegionNPSprctles[4] −0.21
69 = Intensity −0.17
114.872 = Song Length −0.14
22 other features −0.04
10 12 14 16 18
E[f(X)] = 10.801

(a) GB13 Contributions (b) OL Contributions (c) EE Contributions

(d) GB15 Contributions (e) WAO Contributions (f) SV Contributions

Figure 29: Waterfall plot of Shapley values for the test charts.

To start, we have the only difference between GB13 and GB15 being the patterning, so we expect to see one of the pattern-related features to stand out in GB15 and indeed it does. The numerical contribution might be small, but the point is that the model is capable to distinguish between these two near identical chart based on the one differing feature: patterns. We see something similar with OL and WAO, where WAO is just a four-times longer version of OL. The "Song Length" feature shows up in the WAO figure, where it does not in OL. Again, the actual contribution to the prediction might be negligible, but it does show that the model is behaving as intended, which is an important first step.

5.4 Random Forest Classifier

Similar to the random forest regression model, we fine-tune the model parameters like the number of trees, feature selection algorithm and the number of features to consider when looking for best split. However, we treat the output as distinct (albeit sequential) classes, and assess the differences.

5.4.1 Improved Model

For the initial forest, we use as a base again the default set of 29 features as defined in the general methodology section. The modified rest region calculation is used instead of the base case algorithm, and the intensity and total pattern features are added for the improved model. It is then initiated with the following specifications:

Table 18: Initial Model Parameters

Parameter	Value	Clarification
n_estimators	250	Number of trees in the forest.
criterion	"gini"	Using Gini to measure quality of split.
max_depth	None	Max depth of tree. "None" means we expand until all leaves are pure or contain less than the specified minimum samples.
min_samples_split	6	Minimum required samples before splitting a node.
min_samples_leaf	3	Minimum samples required to be at a leaf node.
max_features	None	Number of features to consider when looking for best split.Setting this to "none" means we use a maximum of all features.
oob_Score	True	Whether to use out-of-bag samples to estimate generalization score.

The improved model is built with these parameters.

The random forest regression model is then built in Python using the RandomForestClassifier class from the sklearn.ensemble module. The training data is still the same randomized selection of 90% of our database, and 10% for testing.[19]

5.4.2 Improved Model Results

We finally take note again of the model's general performance:

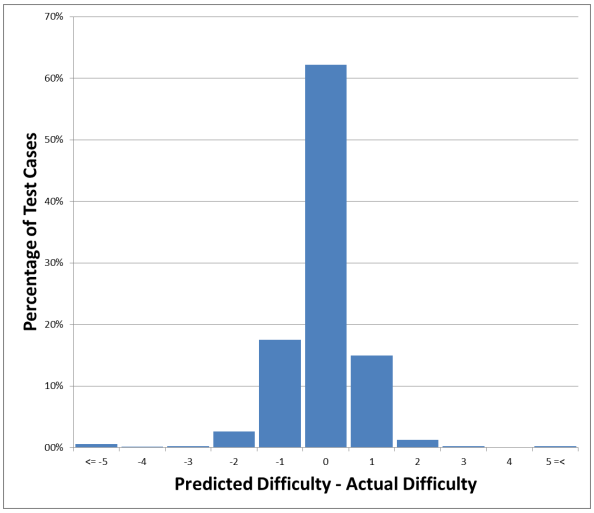
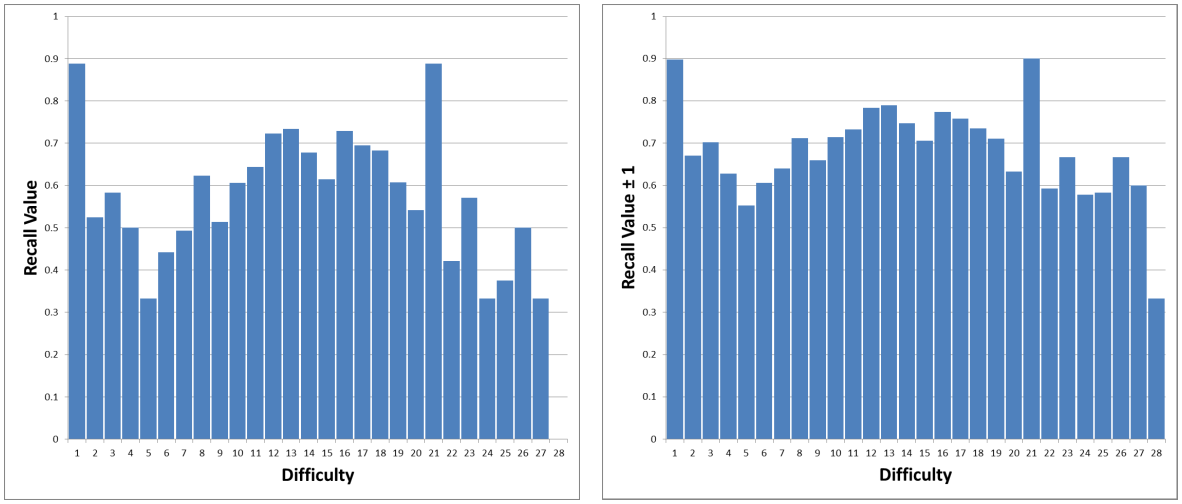


Figure 30: the prediction accuracy of the improved random forest classifier model.

A marginal increase in performance is gained by tweaking the model parameters. This suggest that model choice is generally more important than fine-tuning, at least in our case.A total of 94.6% is identified correctly within a margin of 1. However, the number of charts predicted exactly correct have gone up significantly. This can be made explicit with the classifier-exclusive metric of recall value.

[19]For the code, see main/Random Forest Classifier/RandForest Build.py in the git project.



(a) Recall value per target difficulty. (b) Recall value within an error margin of ±1.

Figure 31: Recall values.

Higher difficulty ratings seem to be recalled relatively better compared to the initial model, but the most striking is that the recall of difficulty 21 specifically has almost doubled. Sudden changes like this are more common in classifier models compared to regression models, as the output is quantized as opposed to continuous.
Next are the feature importance values:

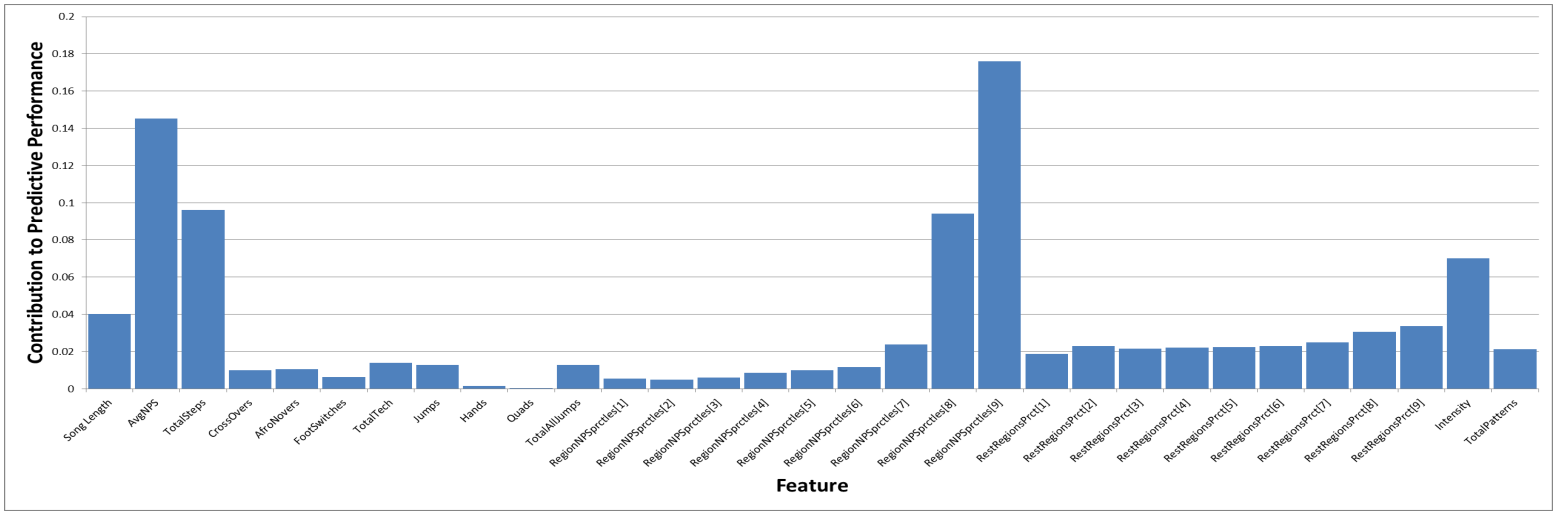
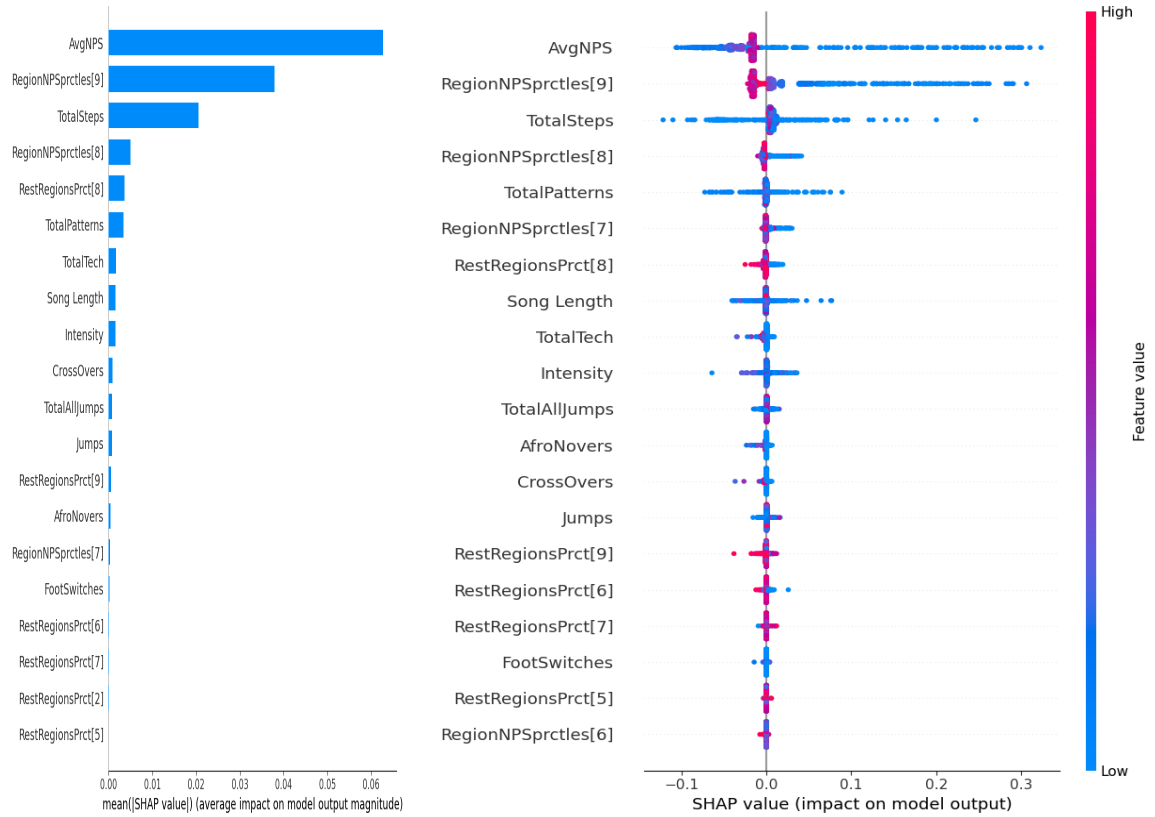


Figure 32: Feature importance of the improved random forest classifier model.

The model displays a very neat distribution of feature importance values, in line with expectations. The new intensity variable seems to play its biggest role yet, despite it not being worked out in detail yet. As this feature is somewhat similar to the idea of path signatures, incorporating those might prove to be very valuable in future research.

Finally, we check the Shapley values once more:



(a) Bar graph.

(b) Scatter-plot.

Figure 33: Shapley values of improved random forest classifier.

The added TotalPatterns and Intensity variables do well as new features, and the contributions of each variable seem to be the least black-box like with the random forest classifier model. Perhaps if some rest regions are to be consolidated into a single feature, the effect might become stronger. As for the base values, ideally these are as low as possible. When inspecting these, they are indeed low but do show an interesting shape:

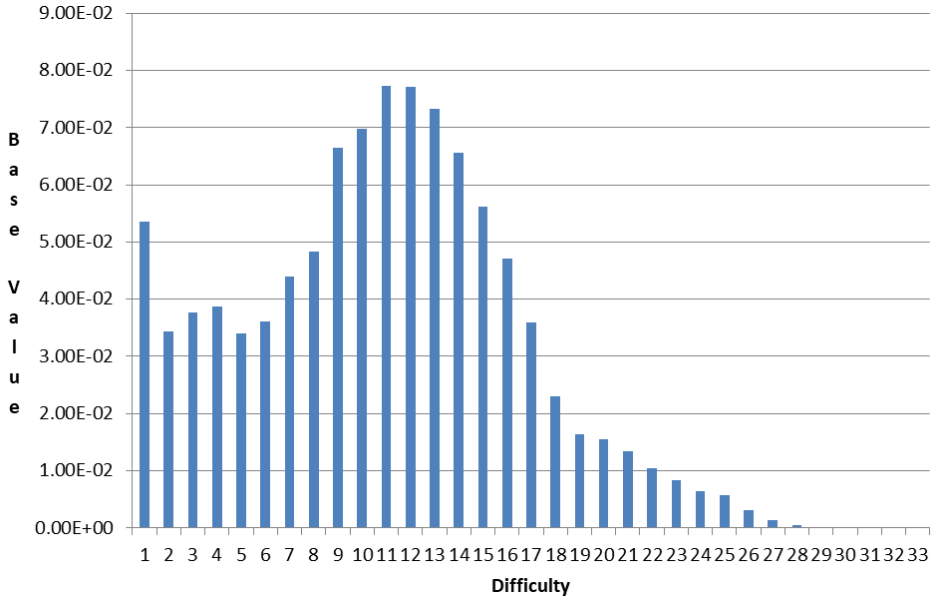


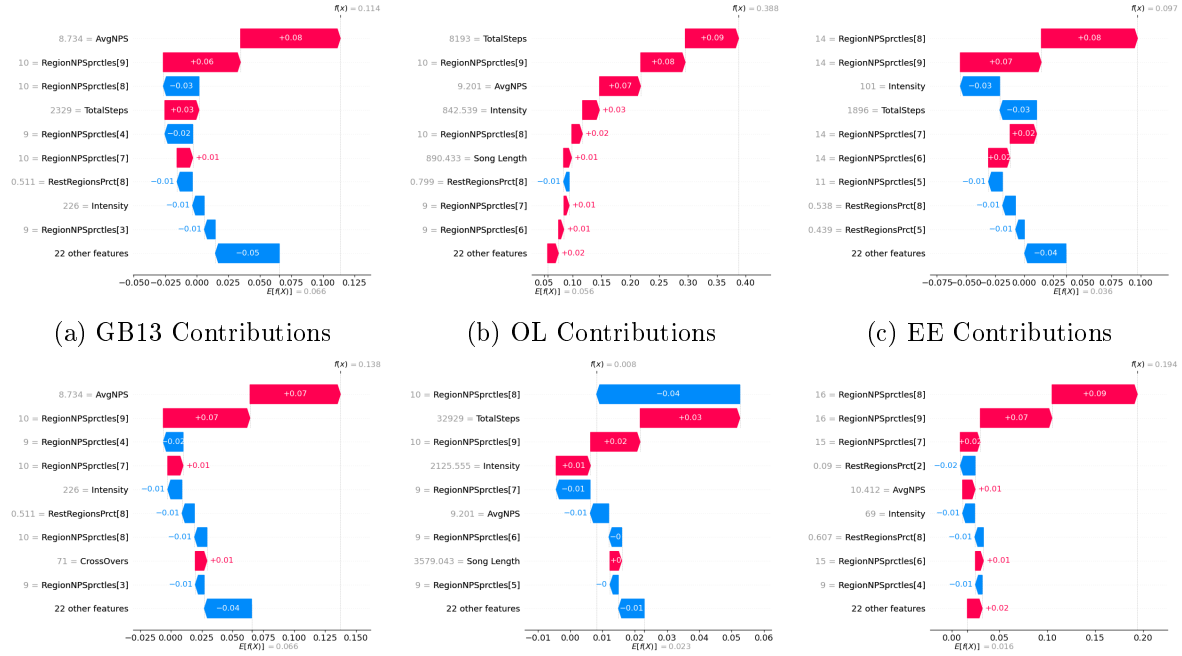
Figure 34: Base values per difficulty.

This is nearly identical to figure 1, indicating a very slight bias towards difficulties appearing more often in the training set. As these values are very low, it doesn't pose a problem but it's an interesting observation nonetheless.
The last remaining performance check is looking at our specific test charts:

Table 19: Random forest classifier predictions on test charts.

Chart Label	Actual Difficulty	Prediction Random Forest Classifier
GB13	13	13
GB15	15	13
OL	14	14
WAO	17	17
EE	16	16
SV	18	18

This is a near perfect score, with the exeption of being able to differ between GB13 and GB15 based on the patterns. We would expect for the features related to patterning to be used for splitting the leaf of those two charts further, but there are several reasons why it might not have. First, the trees might not be deep enough on average, resulting in termination of splitting leaves further before using pattern-related features. Second, the tree-parameters responsible for the minimum required samples before splitting a node and minimum samples required to be at a leaf node might prevent splitting. We tested the latter by rebuilding trees that allow splitting until there's a single sample per leaf. This yielded the same results however. We can look at how the model handles these charts specifically by their Shapley value breakdowns (note that the final value now represents percentage chance of being in a given class, not the predicted difficulty as in the regression case).



(a) GB13 Contributions (b) OL Contributions (c) EE Contributions

(d) GB15 Contributions (e) WAO Contributions (f) SV Contributions

Figure 35: Waterfall plot of Shapley values for the test charts.

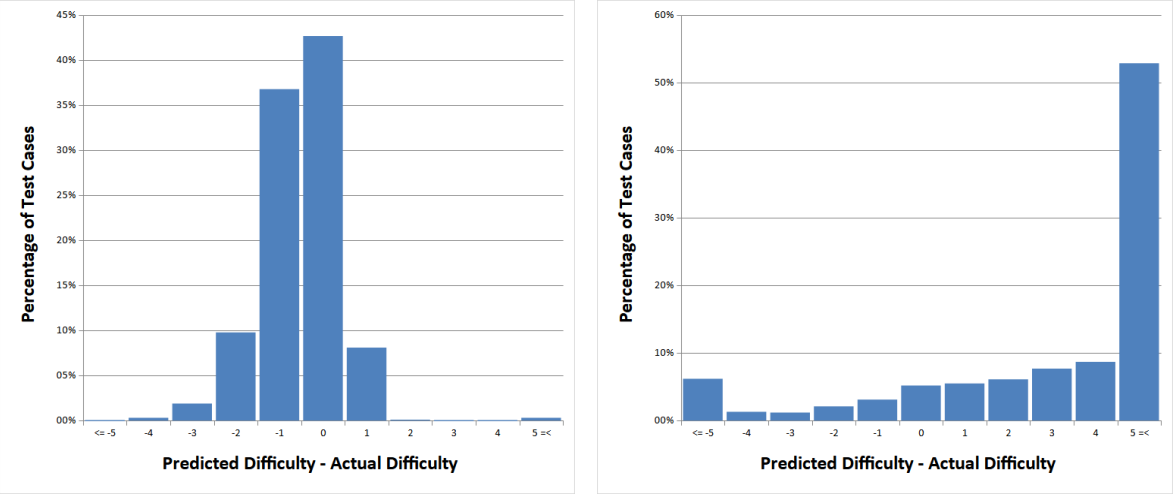
In the case of GB13 and GB15, we do see that the model attributes +1% chance to its certainty about GB15 due to the CrossOvers attribute. This is a specific pattern that occurs only in GB15 and not in GB13. The model might not have predicted the right difficulty for GB15, but it does recognize its distinction to GB13, and clearly falls into a different leaf. The last notable outcome is that there might be over-fitting happening in the case of OL. The model is almost 40% sure about the outcome, which is unusually high. This could be the result of it being in a "lonely leaf" due to its unique characteristics.

5.5 Neural Network

The initial model that was using a classification output layer seemed to be doing informed guesses as shown in the initial model results section. To see whether the regression model works better, we repeat the same input test as before and see whether the model now picks up on patterns in the features.

5.5.1 Improved Model

As seen in figure 36, there is a clear difference between using all features and using no features as inputs.[20]


Percentage of Test Cases
Predicted Difficulty - Actual Difficulty
Percentage of Test Cases
Predicted Difficulty - Actual Difficulty

(a) All features enabled.

(b) No features enabled.

Figure 36: Histogram depicting the prediction accuracy of the regression model input tests.

At this point, the results are getting a lot better. With 87.6% of predicted chart difficulties being within 1 level of the actual difficulty. With the neural network making decent predictions, it is time for hyper-parameter tuning. We do this using a naive cross-validation algorithm. We tune 4 hyper-parameters that we deem to be most important to the overall performance:

Table 20: Tested range of hyper-parameter settings for the improved model.

Hyperparameter	Tested Values
Learning rate	$0.05, 0.15, 0.25, 0.35, 0.45, 0.55$
Activation function	Sigmoid, RELU, smht, tanh
Epochs	$1, 3, 5, 7, 9$
Hidden layer count	1
Neurons per hidden layer	$16, 46, 76, 106, 136, 166, 196, 226, 256$
Used inputs	All $(0 - 27)$

This makes for a total of 1080 different tests.[21]

5.5.2 Improved Model Results

The following tables show the efficiency of certain values for each hyper-parameter. The way this is done is by computing the average loss for that particular value. Note however that the interactions between these hyper-parameters can be complicated, a change in one variable might change how another variable interacts with the whole model. Therefore this data does not fully illustrate the entire range of each setting, but does give some insight into how stable the results are.

[20] All the raw results and graphs of this test can be found in "main/NN/res/inputTests continious error" in the git project.

[21] To reproduce these tests, follow the instructions in "main/NN/CrossValidation.c of the git project.

Table 21: Tested range of hyper-parameter settings for the improved model.

Learning rate values	**Average loss**
0.05	0.007178
0.15	0.006363
0.25	0.006435
0.35	0.006555
0.45	0.006700
0.55	0.210609
Activation functions	**Average loss**
sigmoid	0.000487
RELU	0.025032
tanh	0.134694
Smht (with a = 0.65)	0.002346
Number of epochs	**Average loss**
1	0.032487
3	0.033204
5	0.032755
7	0.058908
9	0.045844
Neurons in first hidden layer values	**Average loss**
16	0.006809
46	0.006630
76	0.043598
106	0.044855
136	0.045373
166	0.046184
196	0.058538
226	0.060901
256	0.052871

The best NN found has a loss of 0.000072. It uses a learning rate of 0.05, the smht function, 9 epochs and had 256 neurons in the first hidden layer. With this, 99.5% of chart difficulty predictions fell within 1 difficulty level of its actual difficulty, and 94.9% of its predictions were exactly equal to the actual difficulty. This is however with a specific train/test split seed, and we might have just gotten lucky. To See how well the model does on average, we use a script that only changes the seed that is used to randomly shuffle the training/testing set such that the initial variables in the neural network model remain exactly the same.[22] The following histogram shows how well the model performs over 100 different seeds:

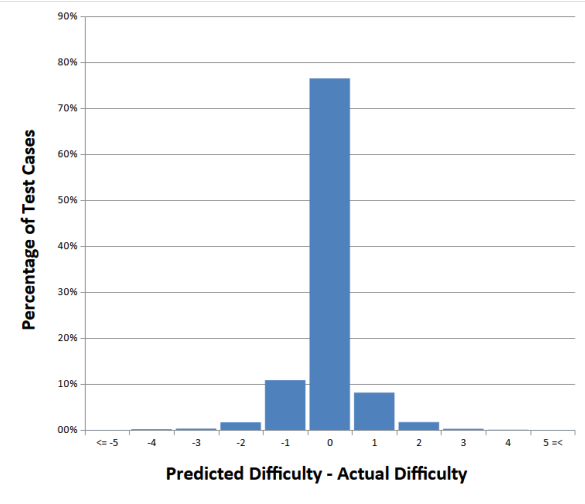


Figure 37: Histogram depicting the prediction accuracy of the improved NN model.

The worst result of these seeds only predicted 40.8% correctly within 1 difficulty level of the actual difficulty. But the best one predicted 100% correctly within the same interval. This tells us that the settings of this model make it fairly unstable. The rounded down median of these 100 seeds was 99% correct predictions within 1 difficulty level of the actual difficulty, which had a prevalence of 75%. This just indicates how important the randomization of the training and testing set is for the overall results.

6 Comparing Performance & Discussion

We now compare the different models and their performance.

Table 22: Predictive performance comparison between models (in percentage of test set).

Model	Correct Predictions (%)	1-off Predictions (%)
Standard Linear Regression	40.45	88.17
Decision Tree	56.40	93.63
Random Forest Regression	63.54	95.24
Random Forest Classifier	62.19	94.60
Neural Network	76.55	95.57

[22] To repeat these builds, follow the instructions written in the comments in main/NN/SeedScript.c of the git project.

	DATASET	SLR	DT	RFR	RFC	NN
GB13	13	13.87	13.50	13.14	13	13.07
GB15	15	14.33	15.00	13.51	13	14.84
OL	14	14.88	15.00	14.43	14	13.98
WAO	17	18.74	16.50	16.97	17	17.02
EE	16	16.94	16.50	16.1	16	16.22
SV	18	18.30	13.17	17.99	18	18.67

Table 23: Predictions of different models for the set of test charts. From left to right, the models are Standard Linear Regression (SLR), Decision Tree (DT), Random Forest Regressio (RFR)n, Random Forest Classifier (RFC) and Neural Network (NN).

It seems like the standard linear regression model performs worst in general, although it does pick up features in the edge-case test charts that some other models don't. In general, the neural network performs best. It has the highest percentage of test cases correct or near-correct, and also did best on the specific chart-tests where it estimated four of the charts exactly correct, and two within a margin of 1. When looking at the percentage of exact correct predictions, it might seem that all models are fairly weak. But remember that the purpose of introducing machine learning into this problem, is that the difficulties are initially assigned subjectively by the creator of a chart. This might very well be slightly off, hence the 1-off prediction paints a much more accurate picture regarding model performance.

7 Conclusion

In our attempt of predicting the difficulty of StepMania game like charts using various machine learning models, we conducted a comprehensive performance comparison. There have been many types of models and ways to initiate and use them, each with their own performance. From the results, it is evident that the neural network performs best with our evaluation metrics. The standard Linear Regression model performs the least favorable overall, albeit demonstrating proficiency in certain edge-case test charts where some other models falter. We note that as the complexity of the models increase, so did their performance. The simple linear regression did the worst, followed by a single decision tree. The more complex random forest models did slightly better, and finally the most complex mode (NN) did best. There is a trade-off however, as the difference in performance between the top three models is marginal but the difference in complexity is significant. We also note that the regression versions of the random forest and neural network did better than their classifier counterparts. In conclusion, while the current models showcase promising predictive performance, ongoing efforts in feature engineering (like using time series analysis and path signatures), model refinement, and database enrichment are required for better accuracy and applicability in predicting chart difficulty.

8 Recommendations

These initial modeling attempts leave room for a lot of improvements. For one, the input space can be much improved upon, by adding and removing features. There's also the possibility to parse charts as a time series, opening up all sorts of time series analysis options like path signatures, ARIMA, ARCH, GARCH, and State Space Models. We have briefly attempted to include path signatures as features in the current models, but did not succeed. However,

the "Intensity" feature was added as a way to see if taking the time-series approach could be beneficial and as the feature proved to be somewhat impactful in all models, that approach does look promising. Especially considering how very crudely the intensity algorithm was implemented, and could be improved upon a lot by changing the parsing script. Adding path signatures would surely prove to be a powerful addition. Additionally, unsupervised learning techniques can be deployed for feature selection, or grouping by similarity. That could effectively create new classes as a feature to be used by the models (marathon class for very long charts, technical class for charts with a lot of technical patterns). Finally, as we've seen that more data improves the quality of the models, the database can be enriched with more charts. There are new ones being created constantly, and only a fraction of all existing charts are in our database. As models are only ever as good as their input, there's progress to be made there as well. Apart from enlarging the database, the quality of the current one can also be improved upon by for example inspecting charts that are consistently heavily mis-rated by the models. These could very well be mislabeled by the creator, leading to a lower quality data set. Removing such instances would improve data-quality and therefore the models.
Then there are the many possible model choices, and corresponding optimizations. Our models are by no means perfectly optimized and fine-tuned, as we only looked at a small selection of meta-analytical data regarding model performance. Not only the initial parameter space can be explored further, but also the train-test split of the database. If by accident some edge cases all happen to be in either the training data or the test data, then the performance show skewed. Then there are the considerations of what percentage of the database to use for testing. Too much and it will limit your training data, too little and the performance tests might not be meaningful. These problems were all taken into account to some level, but there is definitely much more possible. Another improvement to explore would be incorporating models within each other. Outputs of one can be fed into another, creating a more complex modeling infrastructure. But before increasing depth and complexity, it would be a good benchmark test to see how a simple man-made thought out algorithm would perform.
Most of the focus has been on the performance and outcomes of the models, but there are also improvements to be made regarding efficiency. The decision tree initiates several matrices of size 2^{n_layers}, which are per definition at least half empty, as only the last layer of the tree could theoretically need that many columns. Writing better code could significantly reduce memory usage in this case. We also noticed that when severely cutting down on the number of trees in the random forest models or the layers in the decision tree, the performance only suffered minimally, but the time it takes to build and run test cases drastically decrease. If we were to implement one of the models as a plugin for making charts, it must be compact and finish quickly so these are considerations to take into account.